THIRD INTERNATIONAL WORKSHOP ON PARSING TECHNOLOGIES

Sponsored by ACL/SIGPARSE
Association for Computational Linguistics
Special Interest Group on Parsing

Tilburg (The Netherlands)
Durbuy (Belgium)

August 10 – 13, 1993

THIRD INTERNATIONAL WORKSHOP ON PARSING TECHNOLOGIES

Sponsored by ACL/SIGPARSE
Association for Computational Linguistics
Special Interest Group on Parsing

Tilburg (The Netherlands)
Durbuy (Belgium)

August 10 - 13, 1993

McDonald **171**
The Interplay of Syntactic and Semantic Node Labels in Partial Parsing

Nederhof — Sarbo **187**
Increasing the Applicability of LR Parsing

O'Donnell **203**
Reducing Complexity in a Systemic Parser

Oude Luttighuis — Sikkel **219**
Generalized LR Parsing and Attribute Evaluation

Raaijmakers **235**
A Proof-Theoretic Reconstruction of HPSG

Schabes — Waters **257**
Stochastic Lexicalized Context-Free Grammar

Sikkel — op den Akker **267**
Predictive Head-Corner Chart Parsing

Sleator — Temperley **277**
Parsing English with a Link Grammar

Strzalkowski — Scheyen **293**
Evaluation of TTT Parser: A Preliminary Report

Ushioda — Evans — Gibson — Waibel **309**
Frequency Estimation of Verb Subcategorization Frames Based on Syntactic and Multidimensional Statistical Analysis

Weng **319**
Handling Syntactic Extra-Grammaticality

Wittenburg **333**
Adventures in Multi-Dimensional Parsing: Cycles and Disorders

Appendix — *Weerasinghe — Fawcett* **349**
Probabilistic Incremental Parsing in Systemic Functional Grammar

All the presentations in Tilburg will be given in room AZ 9 of the University.

TUESDAY AUGUST 10

08.30 hrs – 09.30 hrs	Registration, Tilburg University, building A, registration desk
09.30 hrs – 09.45 hrs	Opening by Masaru Tomita and Harry Bunt
09.45 hrs – 10.30 hrs	Invited talk: Makato Nagao, Kyoto, Japan. *Varieties of Heurisitcs in Sentence Parsing*
10.30 hrs – 10.50 hrs	coffee/tea
10.50 hrs – 11.25 hrs	Ralph Rönnquist and Mats Wirèn, Saarbrücken, Germany. *Fully Incremental Parsing*
11.25 hrs – 12.00 hrs	Rene Leermakers, Eindhoven, The Netherlands. *The Use of Bunch Notation in Parsing Theory*
12.00 hrs – 12.35 hrs	Daniel Sleator and Davy Temperley, Pittsburgh, USA. *Parsing English with a Link Grammar*
12.35 hrs – 13.40 hrs	lunch
13.40 hrs – 14.15 hrs	Alon Lavie and Masaru Tomita, Pittsburgh, USA. *An Efficient Noise-Skipping Parsing Algorithm for Context-Free Grammars*
14.15 hrs – 14.50 hrs	Fuliang Weng, Las Cruces, USA. *Handling Syntactic Extra-Grammaticality*
14.50 hrs – 15.25 hrs	Paul Oude Luttighuis and Klaas Sikkel, Enschede, The Netherlands. *Generalized LR Parsing and Attribute Evaluation*
15.25 hrs – 15.45 hrs	coffee/tea
15.45 hrs – 16.20 hrs	Mark-Jan Nederhof and J.J. Sarbo, Nijmegen, The Netherlands. *Increasing the Applicability of LR Parsing*
16.20 hrs – 16.55 hrs	Gennaro Costagliola, Pittsburgh, USA. *(Pictorial) LR Parsing from an Arbitrary Starting Point*
16.55 hrs – 17.30 hrs	Hozumi Tanaka, Takenobu Tokunaga and Michio Aizawa, Tokyo, Japan. *Integration of Morphological and Syntactic Analysis Based on the LR Parsing Algorithm*
18.00 hrs	Reception at Tilburg City Hall

Workshop Chairman:	Harry Bunt (ITK, Tilburg)
General Chairman:	Masaru Tomita (CMU, Pittsburgh)
Program Committee:	Robert Berwick Harry Bunt Ken Church Aravind Joshi Ronald Kaplan Martin Kay Bernard Lang Makoto Nagao Anton Nijholt Mark Steedman Henry Thompson Masaru Tomita K. Vijay-Shanker Yorick Wilks Kent Wittenburg
Extra referees	Steven Abney Klaas Sikkel Theo Vosse Paul Oude Luttighuis
Address	Peggy Bertens IWPT'93 Secretariat phone + 31-13-663113 fax + 31-13-662537 email: `iwpt@kub.nl`

Table of Contents

Programme iv

Bod 1
Monte Carlo Parsing

Brill 13
Transformation-Based Error-Driven Parsing

Bunt — van der Sloot 27
Parsing as Dynamic Interpretation

Carpenter 39
Compiling Typed Attribute-Value Logic Grammars

Costagliola 49
(Pictorial) LR Parsing from an Arbitrary Starting Point

Ellis — Garigliano — Morgan 61
A New Transformation into Deterministically Parsable Form for Natural Language Grammars

Garman — Martin — Merlo — Weinberg 73
A Principle-Based Parser for Foreign Language Training in German and Arabic

van der Hoeven 89
An Algorithm for the Construction of Dependency Trees

Hozumi — Takenobu — Michio 101
Integration of Morphological and Syntactic Analysis Based on the LR Parsing Algorithm

Kurohashi — Nagao 111
Structural Disambiguation in Japanese by Evaluating Case Structures based on Examples in Case Frame Dictionary

Lavie — Tomita 123
An Efficient Noise-Skipping Parsing Algorithm for Context-Free Grammars

Leermakers 135
The Use of Bunch Notation in Parsing Theory

Lutz 145
Chart Parsing of Attributed Structure-Sharing Flowgraphs with Tie-Point Relationships

WEDNESDAY AUGUST 11

09.00 hrs – 09.35 hrs	Hideto Tomabechi, Tokushima, Japan. *A Soft Graph Unification Method for Robust Parsing*
09.35 hrs – 10.10 hrs	A. Ruvan Weerasinghe and Robin Fawcett, Cardiff, UK. *Probabilistic Incremental Parsing in Systemic Functional Grammar*
10.10 hrs – 10.45 hrs	Michael O'Donnell, Sydney, Australia. *Reducing Complexity in a Systematic Parser*
10.45 hrs – 11.05 hrs	coffee/tea
11.05 hrs – 11.40 hrs	Klaas Sikkel and Rieks op den Akker, Enschede, The Netherlands. *Predictive Head-Corner Chart Parsing*
11.40 hrs – 12.15 hrs	Bob Carpenter, Pittsburgh, USA. *Compiling of Typed Attribute-Value Logic Grammars*
12.15 hrs – 13.30 hrs	lunch
13.30 hrs – 15.30 hrs	Visit to ITK (Institute for Language Technology and Artificial Intelligence).
15.30 hrs	Bus departure to Durbuy
18.00 hrs	Welcome drinks in conference hotel Le Sanglier des Ardennes, Durbuy

All the presentations in Durbuy will be given in Le Sanglier des Ardennes.

THURSDAY AUGUST 12

08.45 hrs – 09.20 hrs	Nigel Ellis, Roberto Garigliano and Richard Morgan, Durham, UK. *A New Transformation into Deterministically Parsable Form for Natural Language Grammars*
09.20 hrs – 09.55 hrs	James Rogers and K. Vijay-Shanker, Newark, USA. *Towards a Formal Understanding of the Determinism Hypothesis in D-Theory*
09.55 hrs – 10.30 hrs	Stephan Raaijmakers, Tilburg, The Netherlands. *A Proof-Theoretic Reconstruction of HPSG*
10.30 hrs – 10.50 hrs	coffee/tea
10.50 hrs – 11.25 hrs	Tomek Strzalkowksi and Peter Scheyen, New York, USA. *Evaluation of TTT Parser: A Preliminary Report*
11.25 hrs – 12.00 hrs	Gerrit van der Hoeven, Enschede, The Netherlands. *An Algorithm for the Construction of Dependency Trees*
12.00 hrs – 12.35 hrs	Joe Garman, Jeffery Martin, Paola Merlo and Amy Weinberg, Geneve, Switzerland and College Park, Maryland, USA. *A Principle-based Parser for Foreign Language Training in German and Arabic*
12.35 hrs – 14.00 hrs	lunch
14.00 hrs – 15.30 hrs	Discussion time
15.30 hrs – 16.05 hrs	Sadao Kurohashi and Makoto Nagao, Kyoto, Japan. *Structural Disambiguation in Japanese by Evaluating Case Structures based on Examples in Case Frame Dictionary*
16.05 hrs – 16.40 hrs	Eric Brill, Philadelphia, USA. *Transformation-Based Error-Driven Parsing*
16.40 hrs – 16.50 hrs	coffee/tea
16.50 hrs – 17.25 hrs	Harry Bunt and Ko van der Sloot, Tilburg, The Netherlands. *Parsing as Dynamic Interpretation*
17.25 hrs – 18.00 hrs	David McDonald, Brandeis U., USA. *The Interplay of Syntactic and Semantic Node Labels in Partial Parsing*

FRIDAY AUGUST 13

09.00 hrs – 09.35 hrs	Akira Ushioda, Alex Waibel, Ted Gibson and David Evans, Pittsburgh, USA. *Frequency Estimation of Verb Subcategorization Frames Based on Syntactic and Multidimensional Statistical Analysis*
09.35 hrs – 10.10 hrs	Rens Bod, Amsterdam, The Netherlands. *Monte Carlo Parsing*
10.10 hrs – 10.45 hrs	Yves Schabes and Richard Waters, Cambridge, MA, USA. *Stochastic Lexicalized Context-Free Grammar*
10.45 hrs – 11.05 hrs	coffee/tea
11.05 hrs – 11.40 hrs	Rudi Lutz, Brighton, England. *Chart Parsing of Attributed Structure-Sharing Flowgraphs with Tie-Point Relationships*
11.40 hrs – 12.15 hrs	Kent Wittenburg, Bellcore, USA. *Adventures in Multi-Dimentional Parsing: Cycles and Disorders*
12.15 hrs – 12.30 hrs	Closing by Masaru Tomita and Harry Bunt
12.30 hrs – 14.00 hrs	lunch
14.15 hrs	Bus departures to Tilburg and Brussels

Monte Carlo Parsing

Rens Bod

Department of Computational Linguistics, University of Amsterdam
Spuistraat 134, NL-1012 VB AMSTERDAM
email: rens@alf.let.uva.nl

Abstract

In stochastic language processing, we are often interested in the most probable parse of an input string. Since there can be exponentially many parses, comparing all of them is not efficient. The Viterbi algorithm (Viterbi, 1967; Fujisaki et al., 1989) provides a tool to calculate in cubic time the most probable derivation of a string generated by a stochastic context free grammar. However, in stochastic language models that allow a parse tree to be generated by different derivations — like Data Oriented Parsing (DOP) or Stochastic Lexicalized Tree-Adjoining Grammar (SLTAG) — the most probable derivation does not necessarily produce the most probable parse. In such cases, a Viterbi-style optimisation does not seem feasible to calculate the most probable parse. In the present article we show that by incorporating Monte Carlo techniques into a polynomial time parsing algorithm, the maximum probability parse can be estimated as accurately as desired in polynomial time. Monte Carlo parsing is not only relevant to DOP or SLTAG, but also provides for stochastic CFGs an interesting alternative to Viterbi. Unlike the current versions of Viterbi-style optimisation (Fujisaki et al., 1989; Jelinek et al., 1990; Wright et al., 1991), Monte Carlo parsing is not restricted to CFGs in Chomsky Normal Form. For stochastic grammars that are parsable in cubic time, the time complexity of estimating the most probable parse with Monte Carlo turns out to be $O(n^3\varepsilon^{-2})$, where n is the length of the input string and ε the estimation error. In this paper we will treat Monte Carlo parsing first of all in the context of the DOP model, since it is especially here that the number of derivations generating a single tree becomes dramatically large. Finally, a Monte Carlo Chart parser is used to test the DOP model on a set of hand-parsed strings from the Air Travel Information System (ATIS) spoken language corpus. Preliminary experiments indicate 96% test set parsing accuracy.

1 Motivation

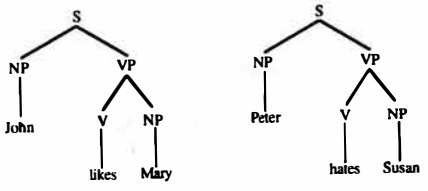
As soon as a formal grammar characterizes a non-trivial part of a natural language, almost every input string of reasonable length gets an unmanageably large number of different analyses. Since most of these analyses are not perceived as plausible by a human language user, there is a need for distinguishing the plausible parse(s) of an input string from the implausible ones. In stochastic language processing, it is assumed that the most plausible parse of an input string is its most probable parse. Most instantiations of this idea estimate the probability of a parse by assigning application probabilities to context free rewrite rules (Jelinek et al., 1990; Black et al., 1992; Briscoe — Carroll, 1993), or by assigning combination probabilities to elementary trees (Resnik, 1992; Schabes, 1992).

There is some agreement now that context free rewrite rules are not adequate for estimating the probability of a parse, since they do not capture lexical context, and hence do not describe how the probability of syntactic structures or lexical items depends on that context. In stochastic lexicalized tree-adjoining grammar (Schabes, 1992), this lack of context-sensitivity is overcome by assigning probabilities to larger structural units. However, it is not always evident which structures should be considered as elementary structures. In (Schabes, 1992), it is proposed to infer a stochas-

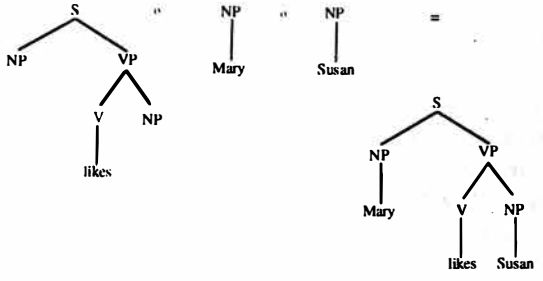
tic TAG from a large training corpus using an inside-outside-like iterative algorithm.

Data Oriented Parsing (DOP) (Scha, 1990,1992; Bod, 1992,1993), distinguishes itself from other statistical approaches in that it omits the step of inferring a grammar from a corpus. Instead, an annotated corpus is directly used as a stochastic grammar. An input string is parsed by combining subtrees from the corpus. In this view, every subtree can be considered as an elementary structure. As a consequence, one parse tree can usually be generated by several derivations that involve different subtrees. This leads to a statistics where the probability of a parse is equal to the sum of the probabilities of all its derivations. It is hoped that this approach can accommodate all statistical properties of a language corpus.

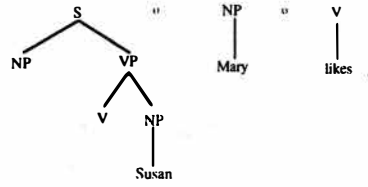
Let us illustrate DOP with an extremely simple example. Suppose that a corpus consists of only two trees:



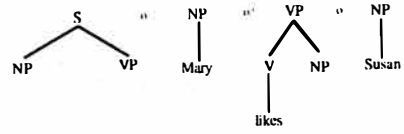
Suppose that our combination operation (indicated with ∘) consists of substituting a subtree on the leftmost identically labeled leaf node of another tree. Then the sentence *Mary likes Susan* can be parsed as an *S* by combining the following subtrees from the corpus. (For an exact definition of subtree, see section 2.)



But the same parse tree can also be derived by combining other corpus subtrees, for instance:



or



Thus, a parse can have several derivations involving different subtrees. These derivations have different probabilities. Using the corpus as our stochastic grammar, we estimate the probability of substituting a certain subtree on a specific node as the probability of selecting this subtree among all subtrees in the corpus that could be substituted on that node. The probability of a derivation can be computed as the product of the probabilities of the subtrees that are combined. As an example, we calculate the probability of the last derivation. The first subtree *S(NP,VP)* occurs twice in the corpus among a total of 20 subtrees rooted with an *S*. Thus, its probability is 2/20. The subtree *NP(Mary)* occurs once among a total of 4 subtrees that can be substituted on an *NP*, hence, its probability is 1/4. The probability of selecting the subtree *VP(V(likes),NP)* is 1/8, since there are 8 subtrees in the corpus rooted with a VP, among which this subtree occurs once. Finally, the probability of selecting *NP(Susan)* is equal to 1/4. The probability of the resulting derivation is then equal to 2/20 * 1/4 * 1/8 * 1/4 = 1/1280. The next table shows the probabilities of the three derivations given above.

P(1st example)	=	1/20 * 1/4 * 1/4	= 1/320
P(2nd example)	=	1/20 * 1/4 * 1/2	= 1/160
P(3rd example)	=	2/20 * 1/4 * 1/8 * 1/4	= 1/1280

This example illustrates that a statistical language model which defines probabilities over parses by taking into account only one derivation, does not accommodate all statistical properties of a language corpus. Instead, we define the probability of a parse as the sum of the probabilities of all its derivations. Finally, the probability of a string is equal to the sum of the probabilities of all its parses.

An important advantage of using a corpus for probability calculation, is that no training of parameters is needed, as is the case for other stochastic grammars (Jelinek et al., 1990; Pereira — Schabes, 1992; Schabes, 1992). Secondly, since we take into account all derivations of a parse, no relationship that might possibly be of statistical interest is ignored. Moreover, this approach does not suffer from a bias in favor of 'smaller' parse trees, as is the case with stochastic CFGs where derivations involving fewer rules, generating 'smaller' trees, are almost always favored regardless of the training material (Magerman — Marcus, 1991; Briscoe — Carroll, 1993). Finally, by using corpus subtrees directly as its structural units, DOP is largely independent of notation systems.

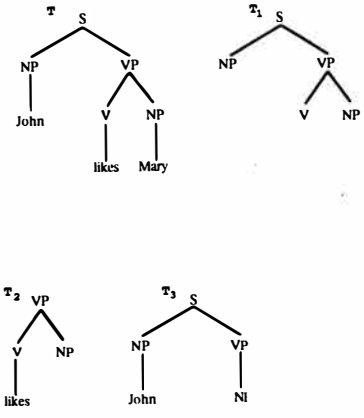
We will show that conventional parsing techniques can be applied to DOP. However, in order to find the most probable parse, a Viterbi-style algorithm does not seem feasible, since the most probable derivation does not necessarily produce the most probable parse. We will show that by using Monte Carlo techniques, the maximum probability parse can be estimated in polynomial time.

In the following, we first outline the DOP model in a more mathematical fashion, and provide an account of Monte Carlo parsing. Finally, we report on some experiments with a Monte Carlo Chart parser on the Air Travel Information System (ATIS) corpus as analyzed in the Penn Treebank.

2 The Data Oriented Parsing Model

A DOP model is characterized by a corpus of tree structures, together with a set of operations that combine subtrees from the corpus into new trees. In this section we explain more precisely what we mean by subtree, operations etc., in order to arrive at definitions of a parse and the probability of a parse with respect to a corpus.

A subtree of a tree T is a connected subgraph S of T such that for every node in S holds that if it has daughter nodes, then these are equal to the daughter nodes of the corresponding node in T. It is trivial to see that a subtree of a tree is also a tree. In the following example T_1 and T_2 are subtrees of T, whereas T_3 isn't.



The definition above also includes subtrees consisting of one node. Since such subtrees do not contribute to the parsing process, we exclude these pathological cases and consider only the set of subtrees consisting of more than one node. We shall use the following notation to indicate that a tree t is a subtree of a tree in a corpus C: $t \varepsilon C =_{\text{def}} \exists T \in C : t$ is a subtree of T, consisting of more than one node.

We will limit ourselves to the basic operation of *substitution*. (Other possible operations which combine subtrees are left to future research.) If t and u are trees, such that the *leftmost non-terminal leaf* of t is equal to the *root* of u, then $t \circ u$ is the tree that results from substituting this non-terminal leaf in t by tree u. The partial function $\circ$ is called *substitution*. We will write $(t \circ u) \circ v$ as $t \circ u \circ v$, and in general $(\ldots((t_1 \circ t_2) \circ t_3) \circ \ldots) \circ t_n$ as $t_1 \circ t_2 \circ t_3 \circ \ldots \circ t_n$.

Tree T is a *parse* of input string s with respect to a corpus C, iff the*yield* of T is equal to s and there are subtrees $t_1, \ldots, t_n \in C$, such that $T = t_1 \circ \ldots \circ t_n$. This definition correctly includes the trivial case of a subtree from the corpus whose yield is equal to the complete input string.

A *derivation* of a parse T with respect to a corpus C is a tuple of subtrees $< t_1, \ldots, t_n >$ such that $t_1, \ldots, t_n \varepsilon C$ and $t_1 \circ \ldots \circ t_n = T$.

Given a subtree $t_1 \varepsilon C$, a function root that yields the root of a tree, and a node labeled X, the conditional probability $P(t = t_1 \mid root(t) = X)$ denotes the probability that t_1 is substituted on X. If $root(t_1) = X$, this probability is 0. If $root(t_1) = X$, this probability can be estimated as the ratio between the number of occurrences of t_1 in C and the total number of occurrences of subtrees t' in C for which holds that $root\ (t\prime) = X$. Evidently, $\sum_i P(t = t_i \mid root(t) = X) = 1$ holds.

The probability of a derivation $< t1, \ldots, tn >$ is equal to the probability that the subtrees $t_1, \ldots, t_n$ are combined. This probability can be computed as the product of the conditional probabilities of the subtrees $t_1, \ldots, t_n$. Let $lnl(x)$ be the leftmost non-terminal leaf of tree x, then:

$$P(< t_1, \ldots, t_n >) = P(t = t_1 \mid root(t) = S) * \prod_{i=2 \text{ to } n} P(t = t_i \mid root(t) = lnl(t_1 \circ \ldots \circ t_{i-1}))$$

The probability of a parse is equal to the probability that any of its derivations occurs. Since the derivations are mutually exclusive, the probability of a parse is the sum of the probabilities of all its derivations. The conditional probability of a parse T given input string s, can be computed as the ratio between the probability of T and the sum of the probabilities of all parses of s.

The probability of a string is equal to the probability that any of its parses occurs. Since the parses are mutually exclusive, the probability of a string s can be computed as the sum of the probabilities of all its parses. It can be shown that $\sum_i P(s_i) = 1$ holds.

3 Monte Carlo Parsing

It is easy to show that in DOP, an input string can be parsed with conventional parsing techniques, by applying subtrees instead of rules to the string (Bod, 1992). Every subtree t can be seen as a production rule $root(t) \rightarrow t$, where the non-terminals of the yield of the right hand side constitute the symbols to which new rules/subtrees are applied. Given a cubic time parsing algorithm, the set of derivations of an input string, and hence the set of parses, can be calculated in cubic time. In order to select the most probable parse, it is not efficient to compare all parses, since there can be exponentially many of them. Although Viterbi's algorithm enables us to derive the most probable derivation in cubic time (Viterbi, 1967; Fujisaki et al., 1989; Wright et al., 1991), this algorithm does not seem feasible for DOP, since the most probable derivation does not necessarily produce the most probable parse. In DOP, a parse can be generated by exponentially many derivations. Thus, even for determining the probability of one parse, it is not efficient to add the probabilities of all derivations of that parse.

It is an open question, whether there exists an adaptation of the Viterbi algorithm that selects the maximum probability parse in cubic time for DOP. In this paper, we pursue an alternative approach. In order to estimate the maximum probability parse efficiently, we will apply Monte Carlo techniques to the decoding problem. We intend to show that, with Monte Carlo, the maximum probability parse can be estimated as accurately as desired, making its error arbitrarily small in polynomial time. Moreover, Monte Carlo techniques can easily be incorporated into virtually any polynomial time parsing algorithm. Thus, Monte Carlo parsing may also provide for stochastic CFGs an interesting alternative to Viterbi, which, in its current versions (Fujisaki et al., 1989; Jelinek et al., 1990; Wright et al., 1991), is restricted to CFGs in Chomsky Normal Form. We will treat Monte Carlo parsing first of all in the context of the DOP model, since it is especially here that the number of derivations generating a single tree

becomes dramatically large.

The essence of Monte Carlo is very simple: it estimates a probability distribution of events by taking random samples (Hammersley — Handscomb, 1964). The larger the samples we take, the higher the reliability. Since the events we are interested in are parses of a certain input string, we should randomly sample parses of that input string. The parse tree which is sampled most often is an estimation of the maximum probability parse. We can estimate the maximum probability parse as accurately as we want by choosing the number of randomly sampled parses as large as we want. The probability of a certain parse T given input string s can be estimated by dividing the number of occurrences of T by the total number of sampled parses N. According to the (Strong) Law of Large Numbers, the estimated probability converges to the actual probability. In the limit of N going to infinity, the estimated probability equals the actual probability: $P(T \mid s) = \#T/N$. From a classical result of probability theory (Chebyshev's inequality) it follows that, independently of the distribution, the time complexity of achieving a maximum estimation error ε by means of random sampling, is equal to $O(\varepsilon^{-2})$.

Let us now turn to the question of how to randomly sample a number of parses of an input string. The most straightforward way seems to be the following: first the set of parses of an input string is derived, yielding a shared parse forest. Next, random samples are taken from this forest, by randomly retrieving parses. Starting for instance at the S-node, a random expansion from the possible expansions is chosen at every node, taking into account the relative frequencies. The parse which is sampled most often is an estimation of the maximum probability parse. Given a cubic time parsing algorithm and assuming that the construction of a parse forest and the retrieval and comparing of parses can be done in cubic time (Leermakers, 1991), the time complexity of this method is $O(n^3\varepsilon^{-2})$ for a string of length n and an estimation error ε.

Depending on the size and the redundancy of the corpus, this method is not always the most efficient one. Instead of applying Monte Carlo techniques after the parsing process, we might also incorporate them *into* the parsing process. This second method consists of calculating a random subset of the parses. Instead of taking into account all candidates[1] at every node in the parsing process, we take a random sample from the total number of candidates at every node. In this way, a set of parses is calculated which is smaller than the total set of parses of an input string. Repeating this process allows us to randomly generate as many parses of a string as desired. If no parses are found during a round, the samples from the candidates may be increased until at least one parse is generated. If, instead, for a new input string a large number of parses is found, the current value of the sample size may be decreased again, and so forth. In the worst case the sample size equals 100% of the total number of candidates and no speedup is achieved. However, this can only happen with non-ambiguous grammars where every string has exactly one derivation. For an ambiguous grammar, any ambiguous string can always be parsed by taking samples from the candidates smaller than the total number of candidates (except that taking a sample from 1 candidate must yield at least that candidate). In our experiments with the ATIS corpus (see next section), it turned out that taking maximally 5% of the candidate subtrees, sufficed to calculate at least one parse for the input string (though often more were found).

As to the time complexity of this second method, it might seem that calculating a subset of exponentially many parses, will yield again exponentially many parses. And comparing exponentially many parses takes exponential time. Nevertheless, by taking the sample sizes relatively small, a tractable upper bound N can be defined, which, if exceeded by the number of parses generated sofar, serves as a stop condition in the repeated parsing process. Secondly, N can be made arbitrarily large, in order to make the estimation error ε arbitrarily small in, as we have seen, quadratic time. Hence, given a cubic time parsing algorithm and assuming that the sample sizes can be made smaller than the total number of candidates but large enough to generate at least one parse (as is the case for redundant grammars like DOP), the time complexity of this method is $O(n^3\varepsilon^{-2})$. Often it suffices to stop repeating the algorithm if the total number of parses ex-

[1] I.e. 'predictions' or 'proposed edges', depending of the kind of parser used.

ceeds a pre-determined bound N. The most frequently generated parse is then an estimation of the maximum probability parse. We shall see in the next section that for the ATIS corpus it sufficed to limit the number of randomly calculated parses to 100, in order to get high parsing accuracy. Though such a small sample may yield inaccurate probabilities for the single parses, it apparently suffices to determine which parse is the most probable one.

Although the worst time complexity of this second method is equivalent to that of the first one, the actual time cost turns out to be much lower. This can be explained by the fact that in the second method only a small part of the actual grammar is used. Since arbitrary CFGs are parsable in $| G |^2$ time, parsing a string 100 times using 5% of the grammar tends to be more efficient than parsing the same string only once using the whole grammar. Secondly, it turns out that the probability estimation of the second method also converges significantly faster. Thus, it seems that this method is especially apt to stochastic parsing with huge amounts of redundant data.

It should be stressed that incorporating Monte Carlo techniques into a parsing algorithm is only feasible if the samples from the candidates can be made much smaller than the total number of candidates, but still large enough to generate at least one parse. Secondly, the demanded maximum error should not be too small, in order to keep the actual time cost to an acceptable degree. For those interested in the Theory of Computation: the algorithms which employ the Monte Carlo techniques described here, are probabilistic algorithms belonging to the class of **B**ounded error **P**robabilistic **P**olynomial time (**BPP**) algorithms. BPP-problems are characterized as follows: it may take exponential time to solve them exactly, but there exists an estimation algorithm with a probability of error that becomes arbitrarily small in polynomial time.

4 Experiments

In order to test the DOP-model, in principle any annotated corpus can be used. This is one of the advantages of DOP: its independence of a notation system. For our experiments[2], we used the naturally occurring Air Travel Information System (ATIS) corpus (Hemphill et al., 1990) as analyzed in the Pennsylvania Treebank (Marcus, 1991; Santorini, 1991). This corpus is of interest since it is used by the DARPA community to evaluate their grammars and speech systems.

We used the standard method of randomly dividing the corpus into a 90% training set and a 10% test set. The 675 trees from the training set were directly used as our stochastic grammar, from which the subtrees and their relative frequencies were derived. The 75 part-of-speech sequences from the test set served as input strings that were parsed with the training set using a Monte Carlo Chart parser (Mijnlief, 1993). To establish the performance of the system, the parsing results were then compared with the trees in the test set. (Note that the "correct" parse was decided beforehand, and not afterwards.)

To measure accuracy, one often uses the notion of *bracketing* accuracy, i.e. the percentage of brackets of the analyses that are not "crossing" the bracketings in the Treebank (Black et al., 1991; Harrison et al., 1991; Pereira — Schabes, 1992; Grishman et al., 1992; Schabes et al., 1993). We believe, however, that the notion of bracketing accuracy is too poor for measuring the performance of a parser. A test set can have a high bracketing accuracy, whereas the percentage of sentences in which no crossing bracket is found (*sentence* accuracy) is extremely low. In (Schabes et al., 1993), it is shown that for sentences of 10 to 20 words (taken from the Wall Street Journal corpus), a bracketing accuracy of 82.5% corresponds to a sentence accuracy of 30%, whereas for sentences of 20 to 30 words a bracketing accuracy of 71.5% corresponds to a sentence accuracy of 6.8%! We shall employ the even stronger notion of *parsing* accuracy, defined as the percentage of the test sentences for which the maximum probability parse is *identical* to the test set parse in the Treebank.

It is one of the most essential features of the DOP approach, that arbitrarily large subtrees are taken into consideration. In order to test the usefulness of this feature, we performed different experiments constraining the *depth* of the subtrees. The depth of a tree is defined as the length of its longest path. The following table shows the results of seven experiments. The accuracy refers to

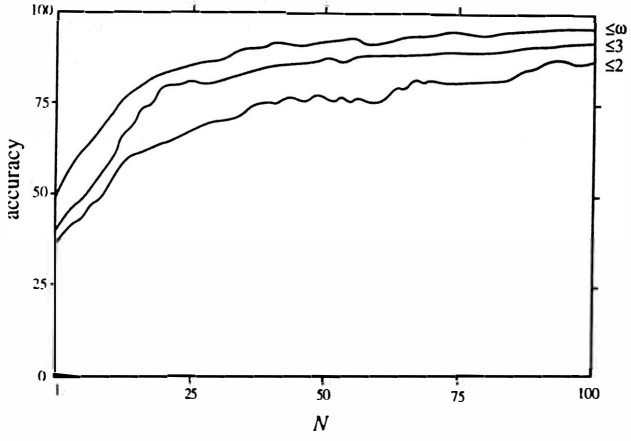
[2]Some of the experiments reported were published in (Bod, 1993).

the parsing accuracy at $N = 100$ sampled parses, and is rounded off to the nearest integer.

depth	accuracy
≤ 2	87%
≤ 3	92%
≤ 4	93%
≤ 5	93%
≤ 6	95%
≤ 7	95%
unbounded	96%

Parsing accuracy for the ATIS corpus, at $N = 100$

The table shows that there is a relatively rapid increase in parsing accuracy when enlarging the maximum depth of the subtrees to 3. The accuracy keeps increasing, at a slower rate, when the depth is enlarged further. The highest accuracy is obtained by using all subtrees from the corpus: 72 out of the 75 sentences from the test set are parsed correctly. In the following figure, parsing accuracy is plotted against the number of randomly generated parses N for three of our experiments: the experiments where the depth of the subtrees is constrained to 2 and 3, and the experiment where the depth is unconstrained.



Parsing accuracy for the ATIS corpus, with depth ≤ 2, with depth ≤ 3 and with unbounded depth

It might also be interesting to look in detail at some parses derived with different constraints on the depths of the subtrees. Consider the test sentence "Arrange the flight code of the flight from Denver to Dallas Worth in descending order", which corresponds to the p-o-s sequence "* VB DT NN NN IN DT NN IN NP TO NP NP IN VBG NN".[3] According to the Treebank, this sentence has the following structure (for a description of the notation system see (Santorini, 1990,1991)):

```
S NP *
  VP VB Arrange
     NP NP DT the
           NN flight
           NN code
        PP IN of
           NP NP DT the
                 NN flight
              PP PP IN from
                    NP NP Denver
                 PP TO to
                    NP NP Dallas
                       NP Worth
     PP IN in
        NP VP VBG descending
           NN order
```

Limiting the depth of the subtrees to 2, the following maximum probability parse was estimated for this string (where for reasons of readability the lexical items are added to the p-o-s tags):

```
S NP *
  VP VB Arrange
     NP NP DT the
           NN flight
           NN code
        PP IN of
           NP NP DT the
                 NN flight
              PP PP IN from
                    NP NP Denver
                 PP TO to
                    NP NP Dallas
                       NP Worth
              PP IN in
                 NP VP VBG descending
                    NN order
```

In this parse tree, we see that the prepositional phrase "in descending order" is incorrectly

[3]Empty elements, like *, had to be treated as part-of-speech elements, in order to be able to use the training set directly as a grammar.

attached to the NP "the flight" instead of to the verb "arrange". This false attachment might be explained by the high relative frequencies of the following subtrees with depth 2 (that appear in structures of sentences like "Show me the transportation from SFO to downtown San Francisco in August", where the PP "in August" is attached to the NP "the transportation", and not to the verb "show").

```
NP NP          NP NP
   PP             PP PP
   PP IN             PP
      NP          PP IN
                     NP
```

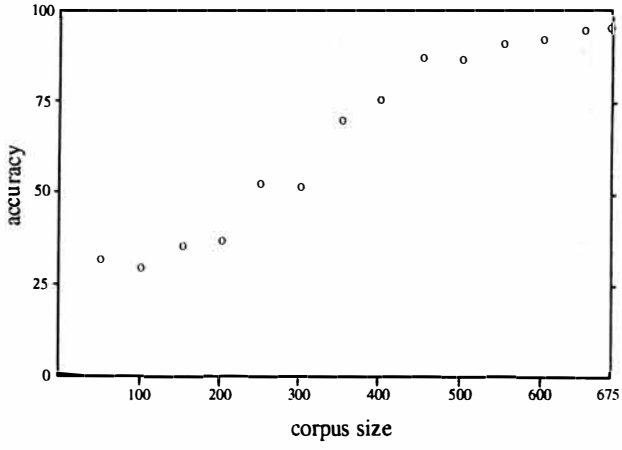
Only if the maximum depth of the subtrees was enlarged to 4, subtrees like the following could be sampled, which led to the estimation of the correct parse tree.

```
VP VB
   NP NP
      PP
   PP IN
      NP VP VBG
         NN
```

It is interesting to note that this subtree occurs only once in the corpus. Nevertheless, it induces the correct parsing of the test sentence. This seems to contradict the observation that probabilities based on sparse data are not reliable (Gale — Church, 1990; Magerman — Marcus, 1991). Since many large subtrees are once-occurring events (hapaxes), there seems to be a preference in DOP for an occurence-based approach if enough context is provided: large subtrees, even if they occur once, tend to contribute to the generation of the correct parse, since they provide much contextual information. Although these subtrees have low probabilities, they tend to induce the correct parse because fewer subtrees are needed to construct a derivation, and therefore the probability of such a derivation tends to be higher than a derivation constructed by many small highly frequent subtrees.

Additional experiments seemed to confirm this hypothesis. Throwing away all hapaxes, yielded an accuracy of 92% (without constraints on the depth of the subtrees and for N = 100), which is a decrease of 4%. Distinguishing between small and large hapaxes, showed that the accuracy was not affected by filtering the subtrees from hapaxes smaller than depth 2 (although the convergence seemed to be slightly faster). Eliminating the hapaxes larger than depth 3, however, decreased the accuracy. Thus, statistical reliability seems only to be relevant if not enough contextual information is available. In such a case, best guesses must be as reliable as possible. When much structural/contextual information is known, on the other hand, there tends to be only one choice. This seems to correspond to the fact that small parts of sentences tend to have many more real structural ambiguities (since not enough information is known) than longer subsentences or whole sentences.

Given the high accuracy achieved by the experiments, we might conclude that the ATIS corpus is a relatively large corpus for its small domain, where almost all relevant constructions occur. It seemed interesting to know how much the accuracy depends on the size of the corpus. For studying this question, we performed additional experiments with different corpus sizes. Starting with a corpus of only 50 parse trees (randomly chosen from the initial training corpus of 675 trees), we increased its size with intervals of 50. As our test set, we took the same 75 p-o-s sequences as used in the previous experiments. In the next figure the parsing accuracy, for $N = 100$, is plotted against the corpus size, using all corpus subtrees.



Parsing accuracy for the ATIS corpus, with unbounded depth.

The figure shows the increase in parsing accuracy. For a corpus size of 450 trees, the accuracy reaches already 88%. After this, the growth decreases, but the accuracy is still growing at corpus size 675. Thus, we might expect an even higher accuracy if the corpus is further enlarged.

Finally, it might be interesting to compare our results with those of others. In (Pereira — Schabes, 1992), 90.36% *bracketing* accuracy was reported using a stochastic CFG trained on bracketings from the ATIS corpus. As said above, the notion of bracketing accuracy is much poorer than that of parsing accuracy. Thus, our pilot experiment suggests that our model has better performance than a stochastic CFG. Some work that achieved high *parsing* accuracy, though with different test data, are the parsers **P**earl and **P**icky of (Magerman — Marcus, 1991) and (Magerman — Weir, 1992). In their work, a stochastic CFG is combined with trigram statistics, yielding about 90% parsing accuracy with word sequences as input strings. We do not yet know what accuracy is achieved if DOP is directly tested on word sequences, instead of on p-o-s sequences. It is likely, that larger corpora are needed for this task.

5 Conclusions

Although a Viterbi-style algorithm provides a tool to derive in cubic time the most probable derivation generated by a stochastic context free grammar, this algorithm does not seem feasible for stochastic language models that allow a parse tree to be generated by different derivations (like DOP or SLTAG), since the most probable derivation does not necessarily produce the most probable parse.

We showed that, by incorporating Monte Carlo techniques into a polynomial parsing algorithm, the most probable parse can be estimated as accurately as desired, making its error arbitrarily small in polynomial time. For stochastic grammars that are parsable in cubic time, the time complexity of estimating the most probable parse with Monte Carlo turns out to be $O(n^3\varepsilon^{-2})$, for a string of length n and an estimation error ε. We suggested that Monte Carlo parsing may also provide for stochastic CFGs an interesting alternative to Viterbi, which, in its current versions, is restricted to CFGs in Chomsky Normal Form. Nevertheless, Monte Carlo parsing seems especially apt to stochastic parsing with huge amounts of redundant data, where one parse is generated by exponentially many (different) derivations.

A Monte Carlo Chart parser was used to test the DOP model on a set of hand-parsed strings from the ATIS corpus. It sufficed to limit the number of randomly calculated parses to 100, in order to get satisfying convergence with high parsing accuracy. It turned out that parsing accuracy improved if larger subtrees were used. Our experiments suggest that statistical reliability is only relevant if not enough structural/contextual information is available.

Acknowledgements

We thank Remko Scha for valuable comments on an earlier version of this paper, and Mitch Marcus for supplying the ATIS corpus.

References

Black E. et al. (1991) "A Procedure for Quantitatively Comparing the Syntactic Coverage of English". In: *Proceedings DARPA Speech and Natural Language Workshop*, Pacific Grove, Morgan Kaufmann.

Black E. — J. Lafferty — S. Roukos (1992) "Development and Evaluation of a Broad-Coverage Probabilistic Grammar of English-Language Computer Manuals". In: *Proceedings ACL'92*, Newark, Delaware.

Bod, R. (1992) "A Computational Model of Language Performance: Data Oriented Parsing". In: *Proceedings COLING'92*, Nantes.

Bod, R. (1993) "Using an Annotated Corpus as a Stochastic Grammar". In: *Proceedings EACL'93*, Utrecht.

Briscoe, T. — J. Carroll (1993) "Generalized Probabilistic LR Parsing of Natural Language (Corpora) with Unification-Based Grammars". In: *Computational Linguistics* 19(1), 25–59.

Fujisaki, T. — F. Jelinek — J. Cocke — E. Black — T. Nishino (1989) "A Probabilistic Method for Sentence Disambiguation". In: *Proceedings 1st Int. Workshop on Parsing Technologies*, Pittsburgh.

Gale, W. — K. Church (1990) "Poor Estimates of Context are Worse than None". In: *Proceedings DARPA Speech and Natural Language Workshop*, Hidden Valley, Morgan Kaufmann.

Grishman, R. — C. Macleod — J. Sterling (1992) "Evaluating Parsing Strategies Using Standardized Parse Files". In: *Proceedings ANLP'92*, Trento.

Hammersley, J. M. — D.C. Handscomb (1964) *Monte Carlo Methods*, Chapman and Hall, London.

Harrison, P. et al. (1991) "Evaluating Syntax Performance of Parser/Grammars". In: *Proceedings of the Natural Language Processing Systems Evaluation Workshop*, Berkeley.

Hemphill C. T. — J.J. Godfrey — G.R. Doddington (1990) "The ATIS spoken language systems pilot corpus". In: *Proceedings DARPA Speech and Natural Language Workshop*, Hidden Valley, Morgan Kaufmann.

Jelinek, F. — J.D. Lafferty — R.L. Mercer (1990) *Basic Methods of Probabilistic Context Free Grammars*, Technical Report IBM RC 16374 (#72684), Yorktown Heights.

Leermakers, R. (1991) "Non-deterministic Recursive Ascent Parsing". In: *Proceedings EACL'91*, Berlin.

Magerman, D. — M. Marcus (1991) "Pearl: A Probabilistic Chart Parser". In: *Proceedings EACL'91*, Berlin.

Magerman, D.— C. Weir (1992) "Efficiency, Robustness and Accuracy in Picky Chart Parsing". In: *Proceedings ACL'92*, Newark, Delaware.

Marcus, M. (1991) "Very Large Annotated Database of American English". *DARPA Speech and Natural Language Workshop*, Pacific Grove, Morgan Kaufmann.

Mijnlief, A. (1993) *A Monte Carlo Chart Parser for DOP*, Dept. of Computational Linguistics, University of Amsterdam.

Pereira, F. — Y. Schabes (1992) "Inside-Outside Reestimation from Partially Bracketed Corpora". In: *Proceedings ACL'92*, Newark.

Resnik, P. (1992) "Probabilistic Tree-Adjoining Grammar as a Framework for Statistical Natural Language Processing". In: *Proceedings COLING'92*, Nantes.

Santorini, B. (1990) *Part-of-Speech Tagging Guidelines for the Penn Treebank Project*, Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia.

Santorini, B. (1991) *Bracketing Guidelines for the Penn Treebank Project*, Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia.

Scha, R. (1990) "Language Theory and Language Technology; Competence and Performance" (in Dutch). In Q.A.M. de Kort & G.L.J. Leerdam (eds.), *Computertoepassingen in de Neerlandistiek*, Almere: Landelijke Vereniging van Neerlandici (LVVN-jaarboek).

Scha, R. (1992) "Virtual Grammars and Creative Algorithms" (in Dutch), *Gramma/TTT* 1(1).

Schabes, Y. (1992) "Stochastic Lexicalized Tree-Adjoining Grammars". In: *Proceedings COLING'92*, Nantes.

Schabes, Y. — M. Roth — R. Osborne (1993) "Parsing the Wall Street Journal with the Inside-Outside Algorithm". In: *Proceedings EACL'93*, Utrecht.

Viterbi, A. (1967) "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". In: *IEEE Trans. Information Theory*, IT-13, 260-269.

Wright, J. — E. Wrigley — R. Sharman (1991) "Adaptive Probabilistic Generalized LR Parsing". In: *Proceedings 2nd Int. Workshop on Parsing Technologies*, Cancun, Mexico.

Transformation-Based Error-Driven Parsing

Eric Brill*

Spoken Language Systems Group
Laboratory for Computer Science, M.I.T.
email: brill@goldilocks.lcs.mit.edu

Abstract

In this paper we describe a new technique for parsing free text: a transformational grammar[1] is automatically learned that is capable of accurately parsing text into binary-branching syntactic trees. The algorithm works by beginning in a very naive state of knowledge about phrase structure. By repeatedly comparing the results of bracketing in the current state to proper bracketing provided in the training corpus, the system learns a set of simple structural transformations that can be applied to reduce the number of errors. After describing the algorithm, we present results and compare these results to other recent results in automatic grammar induction.

1 Introduction

There has been a great deal of interest of late in the automatic induction of natural language grammar. Given the difficulty inherent in manually building a robust parser, along with the availability of large amounts of training material, automatic grammar induction seems like a path worth pursuing. A number of systems have been built that can be trained automatically to bracket text into syntactic constituents. In [MM90] mutual information statistics are extracted from a corpus of text and this information is then used to parse new text. [Sam86] defines a function to score the quality of parse trees, and then uses simulated annealing to heuristically explore the entire space of possible parses for a given sentence. In [BM92a], distributional analysis techniques are applied to a large corpus to learn a context-free grammar.

The most promising results to date have been based on the inside-outside algorithm, which can be used to train stochastic context-free grammars. The inside-outside algorithm is an extension of the finite-state based Hidden Markov Model (by [Bak79]), which has been applied successfully in many areas, including speech recognition and part of speech tagging. A number of recent papers have explored the potential of using the inside-outside algorithm to automatically learn a grammar [LY90, SJM90, PS92, BW92, CC92, SRO93].

Below, we describe a new technique for grammar induction. The algorithm works by beginning in a very naive state of knowledge about phrase structure. By repeatedly comparing the results of parsing in the current state to the proper phrase structure for each sentence in the training corpus, the system learns a set of ordered transformations which can be applied to reduce parsing error. We believe this technique has advantages over other methods of phrase structure induction. Some of the advantages include: the system is very simple, it requires only a very small set of transformations, a high degree of accuracy is achieved, and only a very small training corpus is necessary. The trained transformational parser is completely symbolic and can bracket text in linear time with respect to sentence length. In addition, since some tokens in a sentence are not even

*This work was done while the author was at the University of Pennsylvania. This work was supported by DARPA and AFOSR jointly under grant No. AFOSR-90-0066, and by ARO grant No. DAAL 03-89-C0031 PRI.

[1]Not in the traditional sense of the term.

considered in parsing, the method could prove to be considerably more robust than a CFG-based approach when faced with noise or unfamiliar input. After describing the algorithm, we present results and compare these results to other recent results in automatic phrase structure induction.

2 Transformation-Based Error-Driven Learning

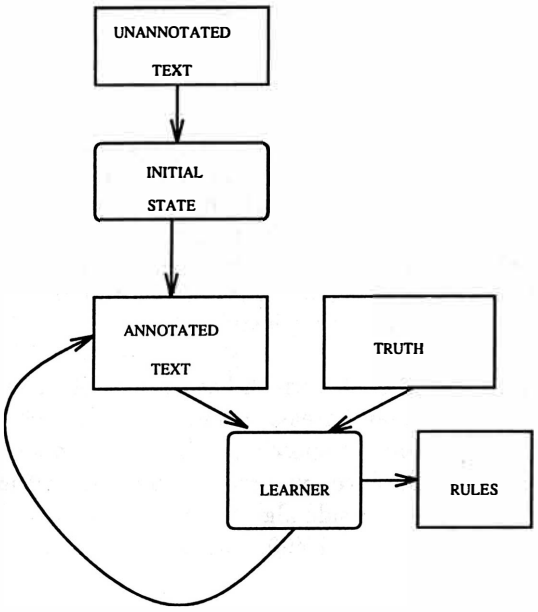


Figure 1: Transformation-Based Error-Driven Learning.

The phrase structure learning algorithm is an application of a general learning technique called transformation-based error-driven learning. This learning paradigm, illustrated in figure 1, has proven to be successful in a number of different natural language applications. In [Bri93] (see also [Bri92, BM92b]), transformation-based learning is applied to part of speech tagging. It is shown that the transformation-based approach outperforms stochastic taggers ([MM91]) when trained on small corpora, and obtains performance comparable to stochastic taggers on larger corpora. This is significant in light of the fact that the transformation-based tagger is completely symbolic. In [BR93], this technique is applied to prepositional phrase attachment. The transformation-based approach is shown to significantly outperform the t-score technique for prepositional phrase attachment described in [HR91].

In its initial state, the transformation-based learner is capable of annotating text but is not very good at doing so. The initial state annotator is typically very easy to create. In part of speech tagging, the initial state annotator assigns every word its most likely tag in isolation, with unknown words being assigned a default tag. In prepositional phrase attachment, the initial state annotator always attaches prepositional phrases low. The naively annotated text is compared to the *true* annotation as indicated by a small manually annotated corpus, and transformations are learned that can be applied to the output of the initial state annotator to make it better resemble the *truth*. The learner learns a set of ordered transformations from a prespecified set of allowable transformations. A greedy search strategy is used to learn transformations: at each stage of learning, the best scoring transformation is learned for whatever scoring function is being used. Four elements must be defined to completely specify a transformation-based learner:

1. The initial-state annotator.
2. The list of allowable transformations.
3. The scoring function.
4. The search strategy.

3 Learning Phrase Structure

The phrase structure learning algorithm is trained on a small corpus of partially bracketed text which is also annotated with part of speech information. All of the experiments presented below were done using the Penn Treebank annotated corpus[MSM93]. The learner begins in a naive initial state, knowing very little about the phrase structure of the target corpus. In particular, all that is initially known is that English tends to be right branching and that final punctuation is final punctuation. Transformations are then learned automatically which transform the output of the naive parser into output which better resembles the phrase structure found in the training corpus. Once a set of transformations

has been learned, the system is capable of taking sentences tagged with parts of speech (either manually tagged text, or the output of an automatic part of speech tagger) and returning a binary-branching structure with nonterminals unlabelled.[2]

3.1 The Initial State Of The Parser

Initially, the parser operates by assigning a right-linear structure to all sentences. The only exception is that final punctuation is attached high. So, the sentence "*The dog and old cat ate .*" would be incorrectly bracketed as:

((The (dog (and (old (cat ate))))) .)

The parser in its initial state will obviously not bracket sentences with great accuracy. In some experiments below, we begin with an even more naive initial state of knowledge: sentences are parsed by assigning them a random binary-branching structure with final punctuation always attached high.

3.2 Structural Transformations

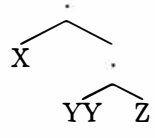
The next stage involves learning a set of transformations that can be applied to the output of the naive parser to make these sentences better conform to the proper structure specified in the training corpus. The list of possible transformation types is prespecified. Transformations involve making a simple change triggered by a simple environment. In the current implementation, there are twelve allowable transformation types:

- (1-8) (*Add*|*delete*) a (*left*|*right*) parenthesis to the (*left*|*right*) of part of speech tag X.
- (9-12) (*Add*|*delete*) a (*left*|*right*) parenthesis between tags X and Y.

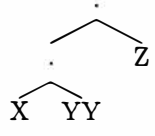
To carry out a transformation by adding or deleting a parenthesis, a number of additional simple changes must take place to preserve balanced parentheses and binary branching. To give an example, to delete a left paren in a particular environment, the following operations take place (assuming, of course, that there is a left paren to delete):

1. Delete the left paren.
2. Delete the right paren that matches the just deleted paren.
3. Add a left paren to the left of the constituent immediately to the left of the deleted left paren.
4. Add a right paren to the right of the constituent immediately to the right of the deleted left paren.
5. If there is no constituent immediately to the right, or none immediately to the left, then the transformation fails to apply.

Structurally, the transformation can be seen as follows. If we wish to delete a left paren to the right of constituent X[3], where X appears in a subtree of the form:



carrying out these operations will transform this subtree into:[4]



Given the sentence:[5]

The dog barked .

this would initially be bracketed by the naive parser as:

[2]This is the same output given by systems described in [MM90, Bri92, PS92, SRO93].

[3]To the right of the rightmost terminal dominated by X if X is a nonterminal.

[4]The twelve transformations can be decomposed into two structural transformations, that shown here and its converse, along with nine triggering environments.

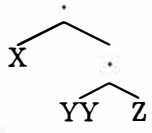
[5]Input sentences are also labelled with parts of speech.

((The (dog barked)) .)

If the transformation *delete a left paren to the right of a determiner* is applied, the structure would be transformed to the correct bracketing:

(((The dog) barked) .)

To add a right parenthesis to the right of YY, YY must once again be in a subtree of the form:



If it is, the following steps are carried out to add the right paren:

1. Add the right paren.
2. Delete the left paren that now matches the newly added paren.
3. Find the right paren that used to match the just deleted paren and delete it.
4. Add a left paren to match the added right paren.

This results in the same structural change as deleting a left paren to the right of X in this particular structure.

Applying the transformation *add a right paren to the right of a noun* to the bracketing:

((The (dog barked)) .)

will once again result in the correct bracketing:

(((The dog) barked) .)

3.3 Learning Transformations

Learning proceeds as follows. Sentences in the training set are first parsed using the naive parser which assigns right linear structure to all sentences, attaching final punctuation high. Next, for each possible instantiation of the twelve transformation templates, that particular transformation is applied to the naively parsed sentences. The resulting structures are then scored using some measure of success that compares these parses to the correct structural descriptions for the sentences provided in the training corpus. The transformation resulting in the best scoring structures then becomes the first transformation of the ordered set of transformations that are to be learned. That transformation is applied to the right-linear structures, and then learning proceeds on the corpus of improved sentence bracketings. The following procedure is carried out repeatedly on the training corpus until no more transformations can be found whose application reduces the error in parsing the training corpus:

1. The best transformation is found for the structures output by the parser in its current state.[6]
2. The transformation is applied to the output resulting from bracketing the corpus using the parser in its current state.
3. This transformation is added to the end of the ordered list of transformations.
4. Go to 1.

After a set of transformations has been learned, it can be used to effectively parse fresh text. To parse fresh text, the text is first naively parsed and then every transformation is applied, in order, to the naively parsed text.

One nice feature of this method is that different measures of bracketing success can be used: learning can proceed in such a way as to try to optimize any specified measure of success. The measure we have chosen for our experiments is the same measure described in [PS92], which is one of the measures that arose out of a parser evaluation workshop [ea91]. The measure is the percentage of constituents (strings of words between

[6] The *state* of the parser is defined as naive initial-state knowledge plus all transformations that currently have been learned.

matching parentheses) from sentences output by our system which do not cross any constituents in the Penn Treebank structural description of the sentence. For example, if our system outputs:

(((The big) (dog ate)) .)

and the Penn Treebank bracketing for this sentence was:

(((The big dog) ate) .)

then the constituent *the big* would be judged correct whereas the constituent *dog ate* would not.

Table 1. shows the first ten transformations found from one run of training on the Wall Street Journal corpus, which was initially bracketed using the right-linear initial-state parser.

#	Add/ Delete	Left/ Right Paren	Environment
1	D	L	Left of NN
2	D	L	Left of NNS
3	A	R	Left of ,
4	D	L	Btwn NNP and NNP
5	D	L	Right of DT
6	A	R	Left of ,
7	D	R	Left of NNS
8	D	R	Btwn NN and NN
9	D	L	Btwn JJ and JJ
10	D	L	Right of $

Table 1: The first 10 learned transformations.

The first two transformations, as well as transformation number 4, 5, 7, 8 and 9 all extract noun phrases from the right linear initial structure. After bracketing in the initial state, every word will be the leftmost terminal of a phrase containing the entire remainder of the sentence to its right. The first two transformations effectively remove singular and plural common nouns from such a structure and bracket them with the preceding constituent instead. The sentence "The cat meowed ." would initially be bracketed as:

((The/DT (cat/NN meowed/VBD)) ./.)

Applying the first transformation to this bracketing (or the second transformation to the same bracketing with *cats* replacing *cat*) would result in:

(((The cat) meowed) .)

If there is a left parenthesis between two proper nouns, then the second proper noun is initially bracketed with constituents that follow it rather than with the preceding proper noun. The fourth transformation fixes this. The sentence *General Motors is very profitable* . would initially be bracketed as:

((General/NNP (Motors/NNP (is (very profitable)))) .)

Applying the fourth transformation would convert this structure to:

(((General Motors) (is (very profitable))) .)

The following example demonstrates the interaction between transformations. The sentence *The fastest cars won* . would initially be bracketed as:

((The/DT (fastest/JJ (cars/NNS won/VBD))) .)

The first transformation to apply to this sentence would be number 2, resulting in:

((The ((fastest cars) won)) .)

The next applicable transformation is number 5, whose application results in:

(((The (fastest cars)) won) .)

After this transformation is applied, no other transformations can be applied to the sentence, and the correct structure is produced.

Transformation number 10 results from the fact that a number usually follows a dollar sign, and these two lexical items should be bracketed together. Transformations 3 and 6 result from the fact that a comma is a good indicator of the preceding phrase being terminated. Since each transformation is carried out only once per environment, multiple listings of a transformation are required if the transformation is to be applied multiple times to a single environment. The sentence *We called them , but they left* . would initially be bracketed as:

((We/PP (called/VBD (them/PP (,/, (but (they left)))))) .)

The first applicable transformation is number 3, whose application results in:

((We ((called them) (, (but (they left))))) .)

The next applicable transformation is number 6, whose application results in the correct structure:

(((We (called them)) (, (but (they left)))) .)

4 Results

In the first experiment we ran, training and testing were done on the Texas Instruments Air Travel Information System (ATIS) corpus[HGD90].[7] In table 2, we compare results we obtained to results cited in [PS92] using the inside-outside algorithm on the same corpus. Accuracy is measured in terms of the percentage of noncrossing constituents in the test corpus, as described above. Our system was tested by using the training set to learn a set of transformations, and then applying these transformations to the test set and scoring the resulting output. In this experiment, 64 transformations were learned (compared with 4095 context-free rules and probabilities used in the inside-outside algorithm experiment). It is significant that we obtained comparable performance using a training corpus only 21% as large as that used to train the inside-outside algorithm.

Method	# of Training Corp Sentences	Accuracy
Inside-Outside	700	90.4%
Transformation Learner	150	91.1%

Table 2: Comparing two learning methods on the ATIS corpus.

After applying all learned transformations to the test corpus, 60% of the sentences had no crossing constituents, 74% had fewer than two crossing constituents, and 85% had fewer than three. The mean sentence length of the test corpus was 11.3. In figure 2, we have graphed percentage correct as a function of the number of transformations that have been applied to the test corpus. As the transformation number increases, overtraining sometimes occurs. In the current implementation of the learner, a transformation is added to the list if it results in *any* positive net change in the training set. Toward the end of the learning procedure, transformations are found that only affect a very small percentage of training sentences. Since small counts are less reliable than large counts, we cannot reliably assume that these transformations will also improve performance in the test corpus. One way around this overtraining would be to set a threshold: specify a minimum level of improvement that must result for a transformation to be learned. Another possibility is to use additional training material to prune the set of learned transformations.

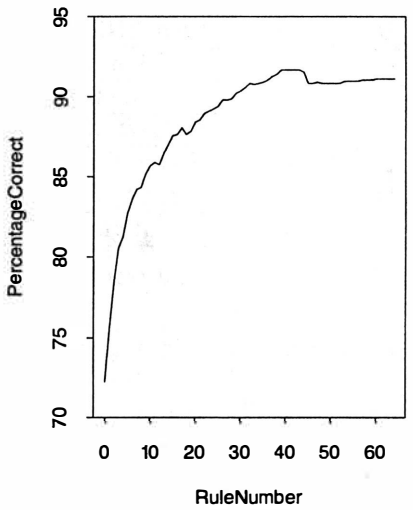


Figure 2: Accuracy as a function of transformation number for the ATIS Corpus.

We next ran an experiment to determine what performance could be achieved if we dropped the initial right-linear assumption. Using the same

[7] In all experiments described in this paper, results are calculated on a test corpus which was not used in any way in either training the learning algorithm or in developing the system.

training and test sets as above, sentences were initially assigned a random binary-branching structure, with final punctuation always attached high. Since there was less regular structure in this case than in the right-linear case, many more transformations were found, 147 transformations in total. When these transformations were applied to the test set, a bracketing accuracy of 87.1% resulted.

The ATIS corpus is structurally fairly regular. To determine how well our algorithm performs on a more complex corpus, we ran experiments on the Wall Street Journal. Results from this experiment can be found in table 3.[8] Accuracy is again measured as the percentage of constituents in the test set which do not cross any Penn Treebank constituents.[9]

Sent. Length	# Training Corpus Sents	# of Trans-formations	% Accuracy
2-15	250	83	88.1
2-15	500	163	89.3
2-15	1000	221	91.6
2-20	250	145	86.2
2-25	250	160	83.8

Table 3: WSJ Sentences

In the corpus we used for the experiments of sentence length 2-15, the mean sentence length was 10.80. In the corpus used for the experiment of sentence length 2-25, the mean length was 16.82. As would be expected, performance degrades somewhat as sentence length increases. In table 4, we show the percentage of sentences in the test corpus that have no crossing constituents, and the percentage that have only a very small number of crossing constituents.[10]

Sent Length	# Training Corpus Sents	% of 0-error Sents	% of ≤2-error Sents
2-15	500	53.7	84.6
2-15	1000	62.4	87.8
2-25	250	29.2	59.9

Table 4: WSJ Sentences

In table 5, we show the standard deviation measured from three different randomly chosen training sets of each sample size and randomly chosen test sets of 500 sentences each, as well as the accuracy as a function of training corpus size for sentences of length 2 to 20.

# Training Corpus Sents	% Correct	Std. Dev.
0	63.0	0.69
10	75.8	2.95
50	82.1	1.94
100	84.7	0.56
250	86.2	0.46
750	87.3	0.61

Table 5: WSJ Sentences of Length 2 to 20.

In [SRO93], an experiment was run using the inside-outside algorithm to train a grammar from the partially bracketed Wall Street Journal corpus. As in the experiment with the ATIS corpus, all possible binary context-free rules were initially allowed, and random probabilities were assigned to each rule. A comparison of this approach to the transformation-based approach is shown in tables 6 and 7. The inside-outside experiment was carried out on sentences of length 1-15, and the transformation-based experiment was carried out on sentences of length 2-15. The inside-outside experiment had a grammar of 4095 probabilistic context free rules, which could be trimmed down to 450 rules without changing performance. 221 symbolic transformations were learned in the transformation-based experiment. In table 6, the

[8] For sentences of length 2-15, the initial right-linear parser achieves 69% accuracy. For sentences of length 2-20, 63% accuracy is achieved and for sentences of length 2-25, accuracy is 59%.

[9] In all of our experiments carried out on the Wall Street Journal, the test set was a randomly selected set of 500 sentences.

[10] For sentences of length 2-15, the initial right linear parser parses 17% of sentences with no crossing errors, 35% with one or fewer errors and 50% with two or fewer. For sentences of length 2-25, 7% of sentences are parsed with no crossing errors, 16% with one or fewer, and 24% with two or fewer.

transformation-based learner is shown to outperform the inside-outside algorithm when parsing accuracy is measured in terms of crossing brackets. In table 7, accuracy is measured as the percentage of sentences with no crossing bracket violations. We believe these results are significant, considering that the transformation-based approach is only a *weakly* statistical learner (only integer addition and comparison is done in learning) and is a completely symbolic parser that can parse in linear time.

Method	# Training Corpus Sents	% Accuracy
Inside-Outside	1095	90.2
Transformation Learner	1000	91.6

Table 6: Comparison of Two Learning Algorithms on the Wall Street Journal: Crossing Bracket Accuracy

Method	# Training Corpus Sents	Sentence Accuracy
Inside-Outside	1095	57.1
Transformation Learner	1000	62.4

Table 7: Comparison of Two Learning Algorithms on the Wall Street Journal: Sentence Accuracy

A graph showing parsing performance for a WSJ run trained on a 500-sentence training corpus (training and testing on sentences of length 2-15) is shown in figure 3. We also ran an experiment on WSJ sentences of length 2-15 starting with random binary-branching structures with final punctuation attached high. In this experiment, 325 transformations were found using a 250-sentence training corpus, and the accuracy resulting from applying these transformations to a test set was 84.7%.

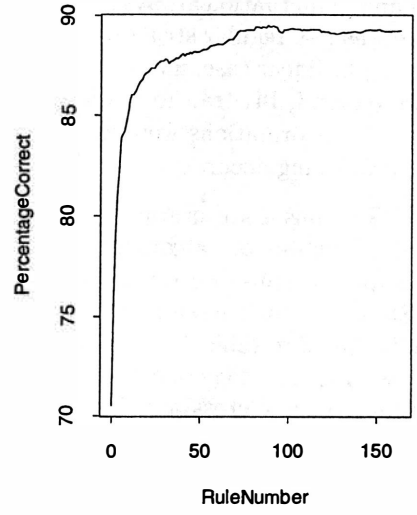


Figure 3: Accuracy as a function of transformation number for the WSJ Corpus.

Finally, in figure 4 we show the sentence length distribution in the Wall Street Journal corpus.

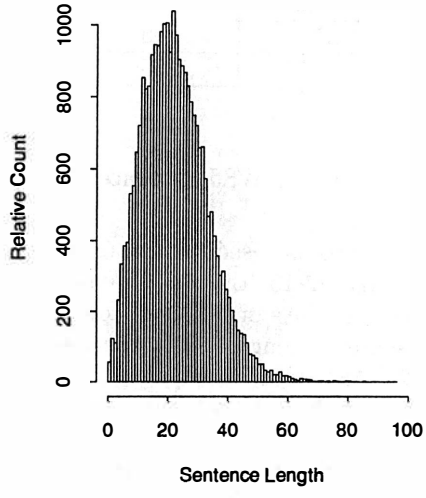


Figure 4: The Distribution of Sentence Lengths in the WSJ Corpus.

While the numbers presented above allow us to compare the transformation learner with systems trained and tested on comparable corpora, these results are all based upon the assumption that the test data is tagged fairly re-

liably (manually tagged text was used in all of these experiments, as well in the experiments of [PS92, SRO93].) When parsing free text, we cannot assume that the text will be tagged with the accuracy of a human annotator. Instead, an automatic tagger would have to be used to first tag the text before parsing. To address this issue, we ran one experiment where we randomly induced a 5% tagging error rate beyond the error rate of the human annotator. Errors were induced in such a way as to preserve the unigram part of speech tag probability distribution in the corpus. The experiment was run for sentences of length 2-15, with a training set of 1000 sentences and a test set of 500 sentences. The resulting bracketing accuracy was 90.1%, compared to 91.6% accuracy when using an unadulterated training corpus. Accuracy only degraded by a small amount when training on the corpus with adulterated part of speech tags.

5 Sample Output

Below are ten randomly chosen parses from the Wall Street Journal. In each case, the output of the bracketing program is listed first, and the Penn Treebank bracketing is listed second. Crossing brackets are marked with a star.

((But (if *((a raider) *(takes *((over (when ((the stock) (is weak)))) (, ((the shareholder) (never (gets (his recovery))))))*)*)*)) .)
((But (if ((a raider) (takes over (when ((the stock) (is weak)))))) , ((the shareholder) (never gets (his recovery)))) .)

(((The company) (expects (to (resume *((full operations) (by today))*)))) .)
(((The company) (expects (to (resume (full operations)) (by today)))) .)

(((" It) *(('s *((very likely) *((((the next) (five years)) (will (be (strong (for funds))))) (, "))*)*) (he says))*) .)
(((" It ('s (very likely ((the next five years) will (be strong (for funds))))) , ") (he says)) .)

(((The (latest report)) (compares (with (((a modest) (9.9 (% increase))) (in *((July (machine orders)) (from ((a year) earlier)))*))))) .)
(((The latest report) (compares (with (a modest 9.9 % increase (in (July machine orders)) (from ((a year) earlier)))))) .)

(((The goal) (was (to (boost ((the circulation) (above ((the (500,000 level)) ((considered significant) (by advertisers))))))))) .)
(((The goal) (was (to (boost (the circulation) (above ((the 500,000 level) (considered significant (by advertisers)))))))) .)

(((Mr. Jones) (ran *((((for (the Senate)) (as (a Democrat))) (in 1986)) (, (but (lost (to ((incumbent Sen.) (Don Nickles)))))))*)) .)
(((Mr. Jones) ((ran (for (the Senate)) (as (a Democrat)) (in 1986)) , but (lost (to (incumbent Sen. Don Nickles))))) .)

((Then (((the ((auto paint) shop)) fire) (sent ((an (evil-looking cloud)) *(of *((black smoke) (into (the air)))*)*)))) .)
((Then ((the auto paint shop fire) (sent (an evil-looking cloud (of (black smoke))) (into (the air))))) .)

((He (*(used *(to (be ((a boiler-room) salesman)))*)* (, (peddling (investments (*(in oil)* *(*((and (gas wells)) and)* (rare coins))*)))))) .)
((He (used (to (be (a boiler-room salesman) , (peddling (investments (in ((oil and gas wells) and (rare coins))))))))) .)

(((The board) (is (scheduled (to (meet Tuesday))))) .)
(((The board) is (scheduled (to (meet Tuesday)))) .)

((Ignore (the (present condition))) .)
((Ignore (the present condition)) .)

In the first example, there are three bracketing errors, all arising from the failure to end the clause following *if* at the comma. The second sentence has one error, which is a prepositional phrase attachment error. The third sentence has three bracketing errors, arising from crossing matching quotes. Perhaps a number of meta-rules, either learned or manually coded,

such as information about matching parentheses and quotes, would significantly improve performance. The fourth sentence has one error, which is again a prepositional phrase attachment error. The sixth sentence has one error, from attaching the clause following (and including) the comma to the preposition *for* instead of the verb *ran*. The seventh sentence has two errors, both due to prepositional phrase attachment. The eighth sentence has five errors, one of which is due to prepositional phrase attachment and two arising from a difficult coordinate structure. In addition to meta-rules, postprocessors addressing particular parsing problems such as prepositional phrase attachment and coordination could lead to significant system performance improvements. Progress has already been made on a transformation-based prepositional phrase attachment program (see [BR93]).

6 Assiging Nonterminal Labels

Once a tree is bracketed, the next step is to label the nonterminal nodes. Transformation-based error-driven learning is once again used for learning how to label nonterminals. Currently, a node is labelled based solely on the labels of its daughters. Therefore, an unlabelled tree can be labelled in a bottom-up fashion. Instead of addressing the problem of labelling the unlabelled tree output of the previous section, we have addressed a slightly different problem. The problem is to assign a tag to a node of a properly bracketed tree given the proper labels for the daughter nodes. This problem can be more easily evaluated and solving it is a significant step toward solving the problem of labelling the output of the transformation-based bracketer.

The Penn Treebank bracketed Wall Street Journal corpus was used for this experiment.[11] Two training sets were used (training set A had 1878 sentences and training set B had 1998), as well as a test set of 1971 sentences. In the first experiment, the initial state annotator assigned the label *noun phrase* to all nodes. Then, transformations were learned to improve accuracy. The transformation templates are:

1. Change the node label to X if Y is a daughter.[12]

2. Change the node label to X if Y and Z are adjacent daughters.

Transformations were learned using training set A. A total of 115 transformations were learned. Initially assigning the label *noun phrase* to all nonterminal nodes in the test set resulted in an accuracy of 44.9%. Applying all learned transformations to the test set resulted in an accuracy of 94.3%. Table 8 shows the first twenty learned transformations. Transformations 15 and 18, as well as a number of similar transformations in the entire list capture the general rule $X \rightarrow$ X and X for coordination. It appears that the transformation *Change a label to S if VP is a daughter* is particularly effective, appearing as transformation 2, 9 and 14. After the second transformation is applied, the transformations that follow could undo the second transformation as a side-effect.

Transformation Number	Tag As	If Daughter Includes
1	PP	IN
2	S	VP
3	VP	VBD
4	VP	VB
5	VP	VBN
6	VP	VBG
7	VP	VBZ
8	S	, S
9	S	VP
10	SBar	-NONE- S
11	PP	TO NP
12	SBar	IN S
13	VP	VBP
14	S	VP
15	S	CC S
16	WHNP	WDT
17	SBar	WHNP
18	VP	CC VP
19	WHNP	WP
20	ADJP	JJR

Table 8: Transformations For Labelling

[11] Thanks to Rich Pito for providing corpus processing tools for running this experiment.

[12] Y can be a nonterminal or preterminal (and need not be the only daughter).

Nonterminals.

So, this transformation applies a number of times to remedy this.

Next, a less naive start state was used. A nonterminal node is assigned the most likely tag for its daughters, as indicated in a second training set (training set B). Unseen daughter sequences are tagged with a default tag (noun phrase). Transformations were learned after applying the start state annotator to training set A. On the test set, initial state accuracy was 92.6%. Applying the transformations resulted in an accuracy of 95.9%. A total of 107 transformations were learned.

We are very encouraged by the accuracy obtained using such a simple learning algorithm that only makes use of very local environments without recourse to any lexical information. Hopefully, adding richer environments such as *the word X is a daughter*, or *the nonterminal to the left is Y* will lead to an even more accurate nonterminal labeller. By first bracketing text and then labelling nonterminals, we can produce labelled parse trees in linear time with respect to sentence length. The bracketer runs in $O(|n| * |T|)$, where $|n|$ is the length of the sentence and $|T|$ is the number of bracketing transformations. The nonterminal labeller also runs in $O(|n| * |T|)$, as all transformations are tried at every nonterminal node. Therefore, parsing run time is: $O(|n| * |T|) + O(|n| * |T|) = O(|n| * |T|)$.

7 Conclusions

In this paper, we have described a new approach for learning a grammar to automatically parse text. The method can be used to obtain high parsing accuracy with a very small training set. Instead of learning a traditional grammar, an ordered set of structural transformations is learned that can be applied to the output of a very naive parser to obtain binary-branching trees with unlabelled nonterminals. Experiments have shown that these parses conform with high accuracy to the structural descriptions specified in a manually annotated corpus. Unlike other recent attempts at automatic grammar induction that rely heavily on statistics both in training and in the resulting grammar, our learner is only very weakly statistical. For training, only integers are needed and the only mathematical operations carried out are integer addition and integer comparison. The resulting grammar is completely symbolic. Unlike learners based on the inside-outside algorithm which attempt to find a grammar to maximize the probability of the training corpus in hope that this grammar will match the grammar that provides the most accurate structural descriptions, the transformation-based learner can readily use any desired success measure in learning.

The transformation-based learner can easily be extended simply by adding transformation templates. In the future, we plan to experiment with other types of transformations. Currently, each transformation in the learned list is only applied once in each appropriate environment. For a transformation to be applied more than once in one environment, it must appear in the transformation list more than once. One possible extension to the set of transformation types would be to allow for transformations of the form: add/delete a paren as many times as is possible in a particular environment. We also plan to experiment with other scoring functions and control strategies for finding transformations and to use this system as a postprocessor to other grammar induction systems, learning transformations to improve their performance. We hope these future paths will lead to a trainable and very accurate parser for free text.

References

[Bak79] J. Baker. Trainable grammars for speech recognition. In *Speech communication papers presented at the 97th Meeting of the Acoustical Society of America*, 1979.

[BM92a] E. Brill and M. Marcus. Automatically acquiring phrase structure using distributional analysis. In *Darpa Workshop on Speech and Natural Language*, Harriman, N.Y., 1992.

[BM92b] E. Brill and M. Marcus. Tagging an unfamiliar text with minimal human supervision. In *Proceedings of the Fall Symposium on Probabilistic Approaches to Natural Language – AAAI Technical Report*. American Association for Artificial Intelligence, 1992.

[BR93] E. Brill and P. Resnik. A transformation based approach to prepositional phrase attachment. Technical report, Department of Computer and Information Science, University of Pennsylvania, 1993. Forthcoming.

[Bri92] E. Brill. A simple rule-based part of speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing, ACL*, Trento, Italy, 1992.

[Bri93] E. Brill. *A Corpus-Based Approach to Language Learning*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1993.

[BW92] T. Briscoe and N. Waegner. Robust stochastic parsing using the inside-outside algorithm. In *Workshop notes from the AAAI Statistically-Based NLP Techniques Workshop*, 1992.

[CC92] G. Carroll and E. Charniak. Learning probabilistic dependency grammars from labelled text – aaai technical report. In *Proceedings of the Fall Symposium on Probabilistic Approaches to Natural Language*. American Association for Artificial Intelligence, 1992.

[ea91] E. Black et al. A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings of Fourth DARPA Speech and Natural Language Workshop*, pages 306–311, 1991.

[HGD90] C. Hemphill, J. Godfrey, and G. Doddington. The ATIS spoken language systems pilot corpus. In *Proceedings of the DARPA Speech and Natural Language Workshop*, 1990.

[HR91] D. Hindle and M. Rooth. Structural ambiguity and lexical relations. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, Berkeley, Ca., 1991.

[LY90] K. Lari and S. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4, 1990.

[MM90] D. Magerman and M. Marcus. Parsing a natural language using mutual information statistics. In *Proceedings, Eighth National Conference on Artificial Intelligence (AAAI 90)*, 1990.

[MM91] R. Weischedel M. Meteer, R. Schwartz. Empirical studies in part of speech labelling. In *Proceedings of the fourth DARPA Workshop on Speech and Natural Language*, 1991.

[MSM93] M. Marcus, B. Santorini, and M. Marcinkiewicz. Building a large annotated corpus of English: the Penn Treebank. To appear in Computational Linguistics, 1993.

[PS92] F. Pereira and Y. Schabes. Inside-outside reestimation from partially bracketed corpora. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, Newark, De., 1992.

[Sam86] G. Sampson. A stochastic approach to parsing. In *Proceedings of COLING 1986*, Bonn, 1986.

[SJM90] R. Sharman, F. Jelinek, and R. Mercer. Generating a grammar for statistical training. In *Proceedings of the 1990 Darpa Speech and Natural Language Workshop*, 1990.

[SRO93] Y. Schabes, M. Roth, and R. Osborne. Parsing the Wall Street Journal with the inside-outside algorithm. In *Proceedings of the 1993 European ACL*, Uterich, The Netherlands, 1993.

Parsing as Dynamic Interpretation

Harry Bunt and Ko van der Sloot

Institute for Language Technology and Artificial Intelligence ITK
P.O.Box 90153, 5000 LE Tilburg, The Netherlands
email: {bunt|sloot}@kub.nl

Abstract

In this paper we consider the merging of the language of feature structures with a formal logical language, and how the semantic definition of the resulting language can be used in parsing.

For the logical language we use the language EL, defined and implemented earlier for computational semantic purposes. To this language we add the basic constructions and operations of feature structures. The extended language we refer to as 'Generalized EL', or 'GEL'. The semantics of EL, and that of its extension GEL, is defined model-theoretically: for each construction of the language, a recursive rule describes how its value can be computed from the values of its constituents. Since GEL talks not only about semantic objects and their relations but also about syntactic concepts, GEL models are nonstandard in containing both kinds of entities.

Whereas phrase-structure rules are traditionally viewed procedurally, as recipes for building phrases, and a rule in the parsing-as-deduction is viewed declaratively, as a proposition which is true when the conditions for building the phrase are satisfied, a rule in GEL is best viewed as a proposition in Dynamic Semantics: it can be evaluated recursively, and evaluates not to true or false, but to the minimal change in the model, needed to make the proposition true.

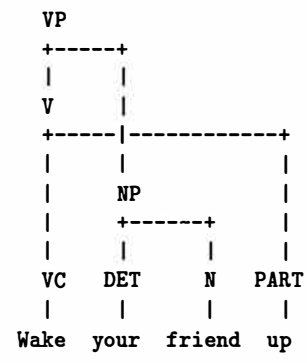
The viability of this idea has been demonstrated by a proof-of-concept implementation for DPSG chart parsing and an emulation of HPSG parsing in the STUF environment.

1 Discontinuous Phrase Structure Grammar

1.1 Introduction

DPSG, or 'Discontinuous Phrase Structure Grammar', has been developed over the years in the context of building natural-language understanding systems, such as the TENDUM dialogue system (Bunt et al., 1985). It has been applied in experimental systems both for parsing and for generation (see Bunt, 1987; 1991; Bunt — Thesingh — Van der Sloot, 1987). It has grown out of the tradition of augmented context-free grammars, where the augmentations in the case of DPSG are (1) conditions on features; (2) the formulation of semantic rules coupled to the syntactic rules in a rule-by-rule fashion.

The distinguishing property of DPSG is that its rules do not construct ordinary phrase-structure trees, but structures that allow crossing branches, called 'discontinuous trees'. Partly inspired by McCawley (1982), these constituents structures have been given a formal definition (Bunt, 1991). The use of discontinuous trees is intended to provide an intuitively appealing and computationally simple treatment of bounded discontinuities. An example is the following:



To generate such a structure DPSG uses rules which, disregarding the augmentations with feature conditions and semantics, look as follows,

where square brackets indicate 'internal context' elements of discontinuous constituents:

```
VP   --> V  + NP
NP   --> DET + N
V    --> VC + [NP] + PART
VC   --> wake
DET  --> your
N    --> friend
PART --> up
```

For further details see Bunt (1991), which also describes a chart parser for DPSG.

1.2 ID-LP separation in DPSG

The above rule format is nowadays obsolete, since it has been recognized that, by separating considerations of immediate dominance (ID) and linear precedence (LP), greater generality can be achieved. This was first demonstrated in GPSG (Gazdar et al., 1985), and has subsequently been exploited to an even greater degree in HPSG (Pollard — Sag, 1987; forthc.). It is, of course, also well recognized in GB theory.

The possibility in principle to apply the ID-LP separation in DPSG, which may be somewhat surprising since a discontinuous PS rule by definition would seem to say something about dominance as well as about precedence, was already indicated in Bunt (1991).

A discontinuous rewrite rule like the one used above — `V --> VC + [NP] + PART` — is equivalent to the following combination of dominance- and precedence constraints, where CD stands for context daughters:[1]

```
ID:   V --> {VC, PART}
CD:   {NP}
LP:   {VC < NP, NP < PART}
```

The dominance and precedence aspects of the rewrite rule have been separated here; one can subsequently consider the possibility to generalize the LP part and make it global for the entire grammar, as GPSG and HPSG are aiming to do, or for a part of the grammar, as is done in the latest version of DPSG.

In this example, the ID/LP formulation clearly has no advantages over the traditional formulation; such advantages can only be expected if the same dominance rule allows various precedences. This is precisely what is often the case with discontinuities; for example, in Dutch an intervening adverb (phrase) can be placed on a variety of positions.

1.3 Semantics in DPSG

In developing DPSG for implementing natural language fragments, we have chosen a rule format somewhat like that of Montague Grammar, where syntactic phrase-structure rules and feature conditions are coupled with semantic rules to build up semantic representations in parallel with syntactic structures.

For these semantic rules we have chosen the language EL ('Ensemble Language'), designed specifically for representing natural language semantics (see Bunt, 1985). This language combines typed lambda abstraction with concepts from Ensemble Theory, an extension of classical set theory developed for dealing with mass terms (Bunt, 1985; see also Lewis, 1992) and with data structures and operations known to be useful from computational semantics, such as lists, list projection, singleton, cartesian product, etc.

2 EL and feature languages

2.1 Feature structures as formal-language expressions

When we regard feature structures, as used for instance in HPSG, as expressions in a formal language, the question arises how this language relates to more traditional formal languages.

Feature structures have three basic ingredients, not commonly found in the formal languages of logic:

1. the pairing of an attribute and a value to form a feature;[2]
2. the combination of features into 'bundles';
3. the use of markers to indicate re-entrancy, or 'structure sharing'.

[1] The rules as given here are, again, extremely oversimplified copared to full-fledged DPSG-rules, which include local and global conditions on features, feature propagation constraints, and semantic composition rules.

[2] The term 'feature' is sometimes used in the literature for the combination of an attribute and a value, like `[case: genitive]` and sometimes for an attribute alone. In this paper we will use the term only in the former sense.

These three ingredients, plus the possibility to use features nested as attribute values, define the core of the language of feature structures. Additional constructions found in feature structures include lists, sets, negation, disjunction and occasionally others. Syntactically, the core of the language of feature structures can be defined as follows.

1. If A is an attribute constant and v a value constant or variable, then $[A : v]$ is a feature expression ("atomic feature specification").
2. If A is an attribute constant and f a feature expression, then $[A : f]$ is a feature expression ("complex feature specification").
3. If f_1 and f_2 are feature expressions, then $[f_1, f_2]$ is a feature expression ("feature bundle").
4. If e is a feature expression and i a structure sharing marker, then $(i : e)$ is a ("marked") feature expression.
5. If A is an attribute constant and i a structure sharing marker, then $[A : i]$ is a feature expression (with structure sharing).

The heart of the corresponding semantic definition is that of the atomic feature specification. We view feature specifications as descriptions of predicates; regarding, for example, the feature specification [gender: neuter] as denoting the predicate of having neuter gender. The semantics of feature bundles is nothing else than the conjunction of the corresponding predicates. The semantics of feature nesting is more complex. To formulate it properly, one must make a 3-way distinction between different kinds of feature attributes, exemplified by the attributes gender, head, and complement-daughters, respectively. Attributes of the first kind, like gender and case, take atomic values to form predicates; one noticeable point is that attributes of the other types cannot have atomic values. Attributes of the second type, like head and synsem, function merely as *labels*, allowing one to refer to certain feature complexes (like the head features) in grammatical principles like the Head Feature Principle. These attributes are semantically vacuous. Attributes of the third kind, like head-daughter and complement-daughters are again different, in that they describe dominance relations between words or phrases, rather than local grammatical properties. This is reflected in the fact that the values of these attributes must refer to words or phrases, i.e. they must be complex feature structures of which the phonology attributes are fully specified.[3] In a model-theoretic semantics, the difference comes out in the fact that these attributes denote relations among the words that inhabit the model and the phrases that can be formed from these words. Such a formalization, which we will not make explicit here, does bring out the asymmetry of HPSG that dominance relations are represented within signs, whereas precedence relations are not. Depending on how far one wants to go in formalizing the language in which the principles of HPSG are formulated, one may want to add the precedence relation to an enriched feature language.

There is one more important aspect of feature structures that must be dealt with: structure sharing. We do this by interpreting structure sharing markers, introduced in the above syntactic rules 4 and 5, as a special kind of variable, and defining the semantics of the expressions in the intuitively obvious way by means of the unification of the values of the subexpressions marked with the sane marker (see also below).

It is clearly possible to add the above five syntactic rules to those of a standard logical language, such as first-order predicate logic. To the rules defining the correct expressions of predicate logic we simply add the above rules and we stipulate that feature specifications can be used everywhere where a one-place predicate constant is allowed. The resulting language can be used to make statements about linguistic material. For instance, the following formula expresses that there is a singular noun with neuter gender in the sentence S :

$$(\exists x) \quad \begin{array}{l} (\mathrm{IN}(x, S) \ \& \\ \mathrm{NOUN}(x) \ \& \\ [\mathrm{number} : \mathrm{sing}, \ \ \mathrm{gender} : \mathrm{neuter}](x)) \end{array}$$

In this way, feature structures can be added to the EL language. This opens the possibility to formulate DPSG rules entirely as expressions in Generalized Ensemble Language (GEL). Evaluating

[3] If empty nodes are allowed, the phonology attribute should obviously be allowed to have an undefined value - which would still make the value of the attribute fully specified.

such GEL-expressions then comes down to trying to prove statements about linguistic material - which is a way of thinking about parsing, as we know from the parsing-as-deduction paradigm.

2.2 Grammatical information in GEL

What do we have to add to EL in order to encode grammatical information? We will consider the case of grammatical information as expressed in Head-driven Phrase Structure Grammar to answer this question.

In HPSG, the sign is the informational unit. Signs make up the lexicon, and are used in the rules of HPSG. A sign is a complex feature structure with certain particular attributes and classes of values; a sign is thus readily expressed in GEL, once the relevant attribute and value constants have been added to the GEL vocabulary.

Rules in HPSG come in three forms: ID schemata, LP restrictions, and general principles (like the Subcategorization Principle and the Head Feature Principle). These rules are all statements about signs.

ID schemata describe, in terms of the various immediate dominance relations distinguished in HPSG (head - head daughter, head - complement daughter, head - adjunct daughter, head - marker, head - filler) the configurations of signs that are permitted. Since the various dominance relations are treated in HPSG as feature attributes, and dominance structures are encoded in feature structures, this means altogether that ID schemata are actually constraints on admissible phrasal signs. Due to the logical power of EL, these constraints are readily expressed in GEL, once we are able to represent signs.

LP restrictions are formally different from ID schemata, since they describe constraints on signs with linear precedence relations, which are not treated in HPSG as feature attributes, and not part of the information in signs, but belong to the metalanguage. For the re-expression of LP restrictions in GEL we have three options:

1. we follow the HPSG strategy, leaving the linear precedence relation at the metalanguage and thus outside the feature formalism;
2. we add a predicate constant denoting the LP relation to GEL, and formalize what the theory says on linear precedence;
3. we introduce a feature attribute 'right neighbour', comparable to HPSG's daughter attributes and treat LP relations as parts of signs.

To make our implementation close to other existing implementations (in particular to the STUF implementation of HPSG, discussed below), we take the first option, although theoretically one could do better.

General principles, finally, describe how the values of certain feature attributes depend on those of the daughters. They can be seen as describing the propagation of feature values upon phrase building. These dependencies can be described in a first-order language, and are thus easily expressed in GEL.

The crucial innovative feature needed in GEL is the possibility to have shared subexpressions within an expression. To this end a new construct has been added which employs special variables ('unifiable variable'), binding shared subexpressions. These variables are denoted by an unique name which starts with '@'. A simple example of a GEL expression with shared subexpressions is:

```
[synsem: [local    : name: @1,
          contents: lambda(_x,
                           application(@1,_x)
                           )]]
```

This could be a template for a propername entry in a lexicon (see below on the definition and use of templates). Whenever @1 is instantiated, this information is shared between the name and the semantic contents. There are no restrictions on the kind of value that can be assigned to a unifiable variable; any GEL expression will do.

An important question is, of course, how this assignment is done. To this end the constructions 'unifiable' and 'unify' are introduced, the use of which will be illustrated below. A unify construction takes any number of arguments, tries to unify them in the usual way (see Shieber et al., 1983), and delivers the resulting GEL expression

or NULL, if the unification fails. Unifiability is defined using unify: unifiable($a_1, \ldots, a_n$) returns TRUE if unify($a_1, \ldots, a_n$) is succesful, and FALSE otherwise.

3 GEL semantics and evaluation

The semantics of EL, and that of its extension GEL, is defined model-theoretically: for each construction of the language there is a rule describing how its value can be computed from the values of its constituent expressions, down to the atomic constituents. Here the recursion ends, and the values of the atomic constituents are looked up in the model, which is a structured specification of these values.

The semantics of GEL is defined model-theoretically, in the same way as that of EL, but since GEL talks not only about 'semantic' objects but also about syntactic objects and their relations, GEL models contain both linguistic and nonlinguistic entities.

When defining a formal semantics for the GEL extension with feature structures, the first thing to consider is the semantics of a simple feature specification. We view a feature attribute like gender as a mathematical function, with entities like feminine as values. This captures the idea that an attribute has a value, and a unique one.

Viewing feature attributes as functions raises the question to what objects these functions apply: what is their domain? We think this is fairly obvious: the domain consists of words and phrases. A feature specification then amounts to a predicate, which can be used to express a syntactic property of a word or phrase. The semantics of a predicate being a set (or a characteristic function), a GEL expression like [gender: fem] receives as its interpretation the set of those words and phrases that have feminin gender.[4]

To formally interpret a feature specification, we apparently need a model which includes words as the 'individuals' to which feature attributes apply, and which also includes syntactic concepts like feminin, interrogative and mass as individual objects that may occur as feature values.

A model for the GEL sublanguage of feature structures is thus a triple:

$$M = < W, \{AV_1, \ldots, AV_k\}, F >$$

where W is a countable set of words, $\{AV_1, .., AV_k\}$ is a finite collection of finite sets of objects called 'atomic feature values', and F is a function assigning interpretations to constants. F assigns atomic feature values to value constants; to an attribute constant A, F assigns a function from W to some value set AV_i. The sets AV_i are assumed to be disjoint.

If $\mathcal{V}$ is the recursive evaluation function assigning interpretations to GEL expressions, we get, in first approximation, the following semantics for a simple feature specification:

$$\mathcal{V}([A : v]) = \{w \text{ in } P(W) | F(A)(w) = \mathcal{V}(v)\};$$
$$\text{if } v \text{ is atomic, then } \mathcal{V}(v) = F(v)$$

where $P(W)$ denotes the set of all phrases (words and word sequences) that can be formed from W.

4 Parsing as dynamic interpretation

Phrase-structure rules are traditionally viewed procedurally, as recipes for building phrases. In the parsing-as-deduction approach a rule is viewed declaratively, as a proposition which is true when the conditions for building the phrase are satisfied. A rule expressed in GEL is best viewed as a proposition in Dynamic Semantics: it can be evaluated recursively, and evaluates not to true or false, but to the minimal change in the model, needed to make the proposition true.

The viability of this idea has been demonstrated by a proof-of-concept implementation for DPSG chart parsing and an emulation of the STUF implementation of HPSG. We describe these in the following sections.

4.1 DPSG rules in GEL

To illustrate the use of GEL in the implementation of DPSG, we consider a (simplified) DPSG rule, which combines a central determiner and a

[4] Treating feature attributes as functions seems to us intuitively more satisfying than treating them as atomic entities, as Gazdar and Pullum (1987) have proposed.

```
NPCENTRE_2
ID rule   : NPCENTRE --> {a:CENTRALDET, b:NOM };
CD rule   : {};
LP constr : {a < b};
GEL rule  : conditional(
                  conjunction( eq( Form_of( a ), Form_of( b ) ),
                               eq( Gender_of( a ), Gender_of( b ) ),
                               memberof( { <Mass>, <Ground>, <Coll> },
                                         Form_of(a) ) ),
                  unify( Head_Feature_Instantiation( b ),
                         Form(    Form_of( b ) ),
                         Gender(  Gender_of( b ) ),
                         Person(  Person_of( b ) ),
                         Content( partselection(
                           Content_of( b), lambda(_x,
                           application( Content_of(a), _x ))) )),
                  undef );
```

nominal into an 'npcentre'. To facilitate the reuse of the parsing strategy implemented for the original DPSG format, the categorial ID-parts as well as the CD- and LP-parts of the rule have been kept separate from the rule part where the real work is done; this part, specifying the conditions and actions on features as well the semantic composition, is expressed in GEL. The GEL expression is of the form `conditional(A, B, C)`; according to the EL semantics underlying GEL, the interpretation of such an expression gives the value of `B` if `A` evaluates to `TRUE`, and that of `C` otherwise. In the conjunction describing the conditions for succesfully applying the rule, the equality relation of EL ('`eq`') is used to test form and gender agreement of the constituents `a` and `b`. In the same way we test whether the form of constituent a is a member of an enumerated set, using the relation '`memberof`'.

When the parameters `a` and `b` in this rule are instantiated by feature structures representing constituents of category `CENTRALDET` and `NOM` (the ID-condition), where the central determiner immediately (CD-condition) precedes the nominal (LP-condition), then the GEL rule says that if the constraints on form and gender are satisfied, these constituents are the daughters of an NP centre whose head features, form feature, gender feature and person feature are inherited (through unification) from the nominal constituent, and whose semantic content is constructed compositionally from the contents of the daughters.[5] Dynamic interpretation of the rule, using the chart as a model, has the effect of *creating* and `NPCENTRE` node with these properties.

This approach has been implemented succesfully by extending the implemented machinery for EL evaluation with the GEL augmentations (basically, the representation of complex feature structures and the operations on those), and merging this with the DPSG parser described in Bunt (1991), originally developed by van der Sloot (1990).

4.2 HPSG/STUF emulated in GEL

4.2.1 The Stuttgart Type Unification Formallism (STUF)

A second proof-of-concept implementation of the idea of parsing as dynamic interpretation has been made in the form of an emulation of the parser and grammar development environment called STUF (Stuttgart Type Unification Formalism), originally developed in the LILOG project (Herzog et al., 1986).

STUF is a descendant of PATR-II (Shieber et al., 1983), providing a formalism and a software environment for writing grammars of various

[5] The semantic part says, more specifically, that the NP centre denotes the union of those parts of the denotation of the (mass) nominal's denotation that satisfy the predicate denoted by the cenral determiner. For more explanation see Bunt, 1985.

kinds. STUF has a much richer language for specifying feature structures, based on Kasper and Rounds' feature logic (Kasper — Rounds, 1986). In STUF it is possible to directly specify complex embedded structures, including disjunctive and negative information. The core of the STUF language is formed by the following definition of the syntax of feature terms:[6]

```
atom              atomic value or template
_X                variable
attribute: S      feature selection
templ(S1,..,Sn)   parametrized template
[S1 S2 ..]        conjunction
{S1 S2 ..}        disjunction
not S             negation
```

Variables are used to describe structure sharing. They have the same interpretation as structure indices in the matrix notation used above.

Templates are named complex feature terms, used to organize information more compactly. An example is:

```
intransitive_verb :=
     [syntax [value: [ syntax: s
                       semantics: _X]
              direction: left
              argument:  syntax: np]
      semantics: _X]
```

Templates may be functional, having parameters which are substituted by actual values when applied. A simple example is:

```
invert_boolean(plus)  := minus.
invert_boolean(minus) := plus.
```

Lexical entries are simply feature term definitions where the name is interpreted as a word that may appear in an input sentence. Templates make these definitions very simple. Examples:

```
John := [ syntax: np ].
goes := intransitive_verb.
```

The STUF implementation of HPSG is not entirely faithful to the theory, in that (1) there is in fact no ID/LP separation in STUF rules, which means that every ID schema has as many LP variants as the LP constraints allow; and (2) the General Principles are instantiated for every ID schema.

The following elements taken from an implementation of a fragment of English in the PLUS system (Black et al., 1991; Rentier, 1993) illustrate the actual use of STUF. In the next subsection we will illustrate the GEL emulation of STUF by describing the corresponding GEL structures.

1. Some functional templates:

```
head(X)   := synsem:local:head: X.
major(X)  := head(maj: X).
inv (X)   := head(maj: X).

comps(X)  := synsem:local:comps: X.
comps0    := comps(undef).

sat       := subj0, comps0.
ssat      := subj0, comps0, mod0.
unsat     := comps(def).

subj(X)   := synsem:local:subj: X.
subj0     := subj(undef).

mod(X)    := synsem:local:mod: X.
mod0      := mod(undef).
```

2. Some nonfunctional templates:

```
nominal   := major(n), mod0.
nphrase   := nominal, ssat.
```

3. Some General Principles and auxiliary templates:

```
'HeadInherit'     := mother: head(X),
                     h_dtr: head(X).
'ContentInherit'  := mother: content(X),
                     h_dtr: content(X).
'CompsInherit'    := mother: comps(X),
                     h_dtr: comps(X).
'GapInherit'      := mother: gap(X),
                     ( ( nonhead: gap0,
                         h_dtr: gap(X) );
                       ( h_dtr: gap0,
                         nonhead: gap(X) ) ).
'BindInherit'     := mother: bind(X),
                     ( ( nonhead: bind0,
                         h_dtr: bind(X) );
                       ( h_dtr: bind0,
                         nonhead: bind(X) ) ).
```

[6] We omit the use of semantic subsorts and paths here. For the original definitions see Dörre — Seiffert, 1991 and Dörre — Raasch, 1991.

```
'NonLocalInherit' :='GapInherit',
                    'BindInherit'.
'ModInherit'      := mother: mod( _X ),
                     h_dtr:  mod( _X ).
'ModCombine'      := mother:( mod0,
                              content(_X),
                              context(_Z) ),
                     h_dtr:    _Y,
                     nonhead:(sat,
                              gap0,
                              mod(_Y),
                              content(_X),
                              context(_Z) ).
'Complementation' := 'HeadInherit',
                     'ModInherit',
                     'ContentInherit',
                     'NonLocalInherit'.
'Adjunction'      := 'HeadInherit',
                     'CompsInherit',
                     'NonLocalInherit',
                     'ModCombine'.
```

4. Some rules (ID schemata with instantiated LP restrictions and General Principles):

```
'RightComplementation' :=
    mother -> h_dtr, nonhead --
    'Complementation',
    nonhead: ( major( not( d ) ) ).
'LeftComplementation'  :=
    mother -> nonhead, h_dtr --
    'Complementation',
    nonhead: ( major(d) ).
'LeftAdjunction'       :=
    mother -> nonhead, h_dtr --
    'Adjunction'.
```

5. Some lexical entries (slightly simplified):

```
car :=  nphrase,
        form(N),
        pers('3rd'),
        parm(P),
        restr1((rel:'CAR', arg0: P)).
who :=  nphrase,
        det( wh ),
        pers('3rd'),
        gend(not(neut)),
        case(not(gen)),
        gap0,
        parm(P),
        restr1((rel:'WHO', arg0: P)).
```

4.2.2 STUF emulated in GEL.

To illustrate the emulation of STUF in GEL, we describe the GEL counterparts of the STUF functional and other templates, ID rules instantiated for LP rules and for General Principles, and a few lexical entries.

0. We first define an empty sign to get unifications going:

```
SIGN := synsem:
          [ local: [ head: [ maj  : @,
                             aux  : @,
                             inv  : @,
                             case : @,
                             form : @ ],
                     comps: @,
                     subj:  @,
                     mod:   @ ],
            nonlocal:
                    [ bind: @,
                      gap:  @ ],
            cont:
                    [ parm :
                            [ pers: @,
                              gend: @,
                              num : @ ],
                      restr: @ ]].
```

The symbol '@' is to be considered as an anonymous variable; no two occurrences are taken as identical.

1. Some functional templates

```
Head( @X )      := synsem:local:head: @X.
Major( @X )     := Head( maj:   @X ).

Inv( @X )       := Head( inv:   @X ).
Inv0            := Inv( undef ).

Comps( @X )     := synsem:local:comps: @X.
Comps0          := Comps( <> ).

Sat             := unify( SIGN,
                          Subj0,
                          Comps0 ).
Ssat            := unify( Sat, Mod0 ).
Unsat           := Comps( Def ).

Subj( @X )      := synsem:local:subj:  @X.
Subj0           := Subj( undef ).

Mod( @X )       := synsem:local:mod: @X.
Mod0            := Mod( undef ).
```

2. Some nonfunctional templates:

```
Nominal := unify( SIGN, Major( N ), Inv0,
                  Mod0, Subj0 ).
Nphrase := unify( Nominal, Ssat ).
```

3. Some General Principles:

```
HeadInherit( @HD ) :=
  conditional( unifiable( @HD, Head(@X) ),
               Head(@X),
               undef ).

ContentInherit( @HD ) :=
  conditional( unifiable( @HD, Content(@X) ),
               Content(@X),
               undef ).

CompsInherit( @HD ) :=
  conditional( unifiable( @HD, Comps(@X) ),
               Comps(@X),
               undef ).

GapInherit(@HD,@N) :=
        conditional(
          unifiable( @N, Gap0 ),
          conditional(
            unifiable( @HD, Gap( @G1 ) ),
            Gap( @G1 ),
            undef ),
          conditional(
            unifiable( @HD, Gap0 ),
            conditional(
              unifiable( @N, Gap( @G2 ) ),
              Gap( @G2 ),
              undef ),
            undef ) ).

BindInherit(@HD,@N) :=
        conditional(
          unifiable( @N, Bind0 ),
          conditional(
            unifiable( @HD, Bind( @G1 ) ),
            Bind( @G1 ),
            undef ),
          conditional(
            unifiable( @HD, Bind0 ),
            conditional(
              unifiable( @N, Bind( @G2 ) ),
              Bind( @G2 ),
              undef ),
            undef ) ).

NonLocalInherit( @HD, @N ) :=
    unify(
        SIGN,
        GapInherit( @HD, @N ),
        BindInherit( @HD, @N ) ).

ModInherit(@HD) :=
  conditional( unifiable( @HD, Mod( @X ) ),
               Mod( @X ),
               undef ).

ModCombine(@HD,@N) :=
        conditional(
          unifiable( @N, Sat,
                     Gap0,
                     Mod(@HD),
                     Content( @X ),
                     Context( @Z ) ),
          unify( SIGN, Mod0,
                       Content( @X ),
                       Context( @Z ) ),
          undef ).

Complementation( @HD, @N ) :=
     unify(
        SIGN,
        HeadInherit( @HD ),
        ModInherit( @HD ),
        ContentInherit( @HD ),
        NonLocalInherit( @HD, @N ) ).

Adjunction( @HD, @N ) :=
    unify(
        SIGN,
        HeadInherit( @HD ),
        SubjInherit( @HD ),
        CompsInherit( @HD ),
        ModCombine( @HD ,@N ) ).
```

4. Some rules (ID schemata with instantiated LP restrictions and General Principles):

```
RightComplementation(@Head,@Comp) :=
    conditional(
      unifiable( @Comp, Major( notu( D ) ) ),
      Complementation( @Head, @Comp ),
      undef ).

LeftComplementation( @Comp, @Head ) :=
    conditional(
      unifiable( @Comp, Major( D ) ),
      Complementation( @Head, @Comp ),
      undef ).

LeftAdjunction( @Adj, @Head ) :=
    Adjunction( @Head, @Adj ).
```

5. Some lexical entries:

```
car := unify(Nphrase,
             Form(N),
             Pers(3rd),
             Parm(@1),
             Restr([rel:Car,
                    argzero:@1])).

who := unify(Nphrase,
             Det(Wh),
             Pers(3rd),
             Gend(notu(Neut)),
             Case(notu(Gen)),
             GapO,
             Parm(@1),
             Restr([rel:Who,
                    argzero:@1])).
```

Comparing the STUF and GEL descriptions, we see that, once the necessary functional templates and other auxiliary structures have been put in place, a relatively simply pattern of relations emerges. It my be noted that the GEL emulation has a clearer and more explicit representation of HPSG's general principles than the original STUF implementation, though the STUF representation is more compact. As a result, the fact that the theory's General Principles are instantiated in every GEL rule (just like in STUF) is hardly a drawback, although it is not in accordance with the theory. The principles are explicitly available in the implementation, ready for inspection and modification. Note also that the lexical representations in STUF and GEL are identical except for minor notational details; this is very important in practice, since all that an HPSG grammar writer is concerned with is the specification of lexical elements. In fact, the correspondence between lexical items in STUF and GEL is so straightforward that an automatic conversion from one format to the other would be possible.

4.3 Rule application as evaluation

Since the LP restrictions and General Principles are instantiated in each ID-rule, application of STUF HPSG rules simply comes down to the application of these ID rules; there are no additional checks. Therefore, standard parsing procedures can be applied. We have implemented a simple chart parser, using the matrix-driven parsing strategy described in Bunt (1991),[7] which applies the ID rules by invoking the implemented 'GEL machine' to evaluate the GEL rule.

5 Conclusions and future work

Where STUF and similar formalisms and implementations developed in recent years, such as TFS, CUF and PLEUK, is an advancement compared to PATR-II because of its more powerful language for expressing linguistic information, GEL offers further extended possibilities to formalize grammatical information and represent it in a computationally attractive form. Not only ID-schemata and lexical entries can be given a formal representation, but LP-restrictions and General Principles as well. At the same time, GEL retains the advantages of a fully declarative representation and of integrated representation of syntactic and semantic information.

Further work will make clear how attractive this approach can be for building efficient parsers and generators that work directly on the constraint-based representation of rules, rather than by internally compiling them first into more traditional formats.

[7] The implemented parser is simplified in that the particular provisions for dealing with discontinuous constituents have for the moment been left out. The implementation, done in C, is the work of Ko van der Sloot.

References

Black, W. et al. (1991) "A Pragmatics-vased Language Undrstanding System". In: *Information Processing Systems and Software: Results of Selected Projects.* EC, Esprit, Brussels.

Bunt, H. (1985) *Mass terms and model-theoretic semantics.* Cambridge University Press, Cambridge, England.

Bunt, H. (1987) "Utterance generation from semantic representation augmented with pragmatic information". In G. Kempen (ed.) *Natural language generation.* Kluwer/Nijhoff, The Hague.

Bunt, H. (1991) "Parsing with Discontinuous Phrase Structure Grammar". In M. Tomita (ed.) *Current Issues in Parsing Technology.* Kluwer, Boston.

Bunt, H. — J. Thesingh — K. van der Sloot (1987) "Discontinuities in trees, rules and parsing". In *Proceedings of the Third Conference of the European Chapter of ACL*, Copenhagen.

Dörre, J. — I. Raasch, (1991) *The Stuttgart Type Unification Formalism - User Manual.* IWBS Report 168, IBM Scientific Center, Stuttgart.

Dörre, J. — R. Seiffert (1991) *Sorted Feature Terms and Relational Dependencies.* IWBS Report 153, IBM Scientific Center, Stuttgart.

Gazdar, G. — E. Klein. — G. Pullum — I. Sag (1985) *Generalized Phrase-Structure Grammar.* Harvard University Press, Cambridge, MA.

Gazdar, G. — G. Pullum (1987) *A Logic or Category Definition.* Cognitive Science Research Paper CSRP 072, University of Sussex.

Herzog, O. et al. (1986) *LILOG - Linguistic and logic methods for the computational understanding of German.* LILOG Report 1b, IBM Scientific Center, Stuttgart.

Kasper, R. — W. Rounds (1986) "A logical semantics for feature structures". In: *Proceedings of the 24th Annual Meeting of the ACL.* Columbia University, New York.

Lewis, D. (1992) *Parts of Classes.* Basil Blackwell, Oxford and Cambridge, MA.

J. McCawley (1982) "Parentheticals and Discontinuous Constituent Structure". *Linguistic Inquiry*, 13, 91-106

Pollard, C. — I. Sag (1987) *Information-based Syntax and Semantics.* Vol I, Fundamentals. CSLI Lecture Notes 13, Center for the Study of Language and Information, Stanford.

Pollard, C. — I. Sag (forthc.) *Information-based Syntax and Semantics.* Vol II. Preliminary version dstributed as 'Topics in Constraint-based Syntactic Theory', Universität des Saarlandes, Saarbrücken, 1992.

Rentier, G. (1993) "The PLUS Grammar". *PLUS deliverable D3.2*, ITK, Tilburg, May 1993.

Shieber, S. — H. Uszkoreit — F. Pereira — J. Robinso — M. Tyson (1983) "The formalism and implementation of PATR-II". In: J. Bresnan (ed.) *Research on Interactive Acquisition and Use of Knowledge.* SRI International, Artificial Intelligence Center, Menlo Park, Cal.

Sloot, K. van der (1990) "The TENDUM 2.7 parsing algorithm for DPSG". *ITK Research Memo*, ITK, Tilburg.

Tomita, M.(ed.) (1991) *Current Issues in Parsing Technology.* Kluwer, Boston.

Compiling Typed Attribute-Value Logic Grammars

Bob Carpenter

Computational Linguistics Program, Philosophy Department
Carnegie Mellon University, Pittsburgh, PA 15213
email: carp@lcl.cmu.edu

Abstract

The unification-based approach to processing attribute-value logic grammars, similar to Prolog *interpretation*, has become the standard. We propose an alternative, embodied in the Attribute-Logic Engine (ALE) (Carpenter 1993), based on the Warren Abstract Machine (WAM) approach to *compiling* Prolog (Aït-Kaci 1991). Phrase structure grammars with procedural attachments, similar to Definite Clause Grammars (DCG) (Pereira — Warren 1980), are specified using a typed version of Rounds-Kasper logic (Carpenter 1992). We argue for the benefits of a strong and total version of typing in terms of both clarity and efficiency. Finally, we discuss the compilation of grammars into a few efficient low-level instructions for the basic feature structure operations.

1 Compiling Type Definitions

The first component of an ALE grammar is a type specification, which lays out the basic types of feature structures that will be employed in a grammar, along with the inheritance relations between these types and declarations of appropriate features and constraints on their values. Such a specification includes declarations such as the following for lists of atoms:

```
bot sub [atom,list].
  atom sub [a,b].
    a sub [].
    b sub [].
  list sub [ne_list,e_list].
    e_list sub [].
    ne_list sub []
            intro [hd:atom,tl:list].
```

The idea here is that bot is the most general type, with two subtypes atom and list. The type atom has two subtypes, a and b, which are maximally specific types. The list type also has two subtypes, ne_list and e_list for non-empty and empty lists, respectively. Note that the ne_list type introduces two features, hd and tl, whose values are required to be atoms and lists. The idea here is that the only type which has any appropriate features is the ne_list type, and it is appropriate for exactly two features, hd and tl.

Inheritance of appropriateness specifications is performed on the basis of the type hierarchy. For instance, consider the following declaration from HPSG:

```
sign sub [word,phrase]
     intro [phon:phon_list,
            synsem:synsem_obj,
            qstore:quant_list].
  word sub []
       intro [phon:singleton_phon_list].
  phrase sub []
         intro [dtrs:dtr_struct].
```

Here the type sign introduces three features and provides value restrictions. The subtype for words inherits these features and the associated value restrictions, imposing the additional condition that the phonology value be a singleton list. In addition, the subtype for phrases introduces an additional features for daughters, which is only appropriate for phrases. Thus, unlike the case for order-sorted terms (see, for instance, Meseguer et al. (1987)), not every subtype of a type need have the same slots for values. This is significant in terms of implementations, as memory cells are

only allocated on a structure for appropriate features.

The initial stage of compilation in ALE involves just the type hierarchy. First, the transitive closure of subsumption is calculated using Warshall's algorithm (see O'Keefe (1990)). Second, least upper bounds are computed for each pair of consistent types. A condition on type hierarchies is that they form a bounded-complete partial order (BCPO), or in other words, that every pair of bounded (consistent) types, those pairs of types with a common subtype, has a least upper bound. This ensures that the unification of two types always takes a unique value. This reduces non-determinism at run-time, but might require additional types to be declared by the user (see Carpenter (1992)). Such hierarchies can be compiled automatically from either systemic networks or ISA/ISNOTA hierarchies, as shown in (Carpenter — Pollard 1991), and such a compiler has been developed and will be included in the next release of ALE (Carpenter and Penn forthcoming). The final stage involves calculating which features are appropriate for each type and their appropriate values. This is done by collecting all of the declared features on subtypes and unifying their value restrictions. The second condition on type hierarchies, in addition to their forming a BCPO, is that they introduce each feature at a unique most general type. This, along with the BCPO condition, ensures a unique solution to the type inference problem. If we only knew that a feature `f` was defined, and nothing else about an object, then if there were two maximally general types for which `f` was appropriate, a decision could not be made as to which type it was and non-determinism would be introduced. As with the BCPO condition, this condition can be automatically eliminated by introducing a new type appropriate for the feature which is more general than the two existing ones (see Carpenter (1992)). We also forbid appropriateness cycles such as:

```
person sub [male,female]
       intro [father:male,
              mother:female].
  male sub [].
  female sub [].
```

We rule out this situation because type inference, as we define it below, can not find most general well-typings in such cases. To be a well-formed object of type `male`, a requirement is that the `father` feature is defined and filled by another male, leading to a non-halting procedure. Again, if we wish to represent people with parents, the problem can be solved by adding types which are not required to have parents.

During the compilation of the type system, many different kinds of errors are detected, such as: two types which mutually subsume one another, violations of the BCPO condition where two types have multiple unifiers, cases where inconsistent constraints are inherited by a feature, where there are appropriateness cycles, where there is no most general type appropriate for a feature, and so on. Other errors such as undeclared types and multiple declarations are also recognized. In addition, a number of warnings are raised in cases which might not be desirable, such as a type with only one subtype or where dynamic type-inference during unification will be necessary. This latter condition arises when types `s` and `t` are both appropriate for `f`, with value restriction `s'` and `t'`, but the unification of `s` and `t`, `s+t`, has a more specific restriction than `s'+t'`. In this case, when an `s` and `t` object are unified, additional constraints on their value for a feature must be checked.

2 Compiling Basic Operations

As with other grammar formalisms based on attribute-value logics, the primary data structure used in ALE is the feature structure. The structures used in ALE are similar to those in other systems, with the primary difference being that they are required to be totally well-typed (see Carpenter (1992)). In other words, every feature structure must be assigned a type and every feature appropriate for that type must appear with an appropriate value. This can be contrasted with sorted, but untyped systems, which allow sorts to label feature structures and participate in unification, but don't enforce any typing conditions. It can also be contrasted with systems which only perform type inference on values, but do not require every appropriate feature to be present. There are a number of benefits to typing a programming language. Not the least of these benefits is the ability to detect errors at

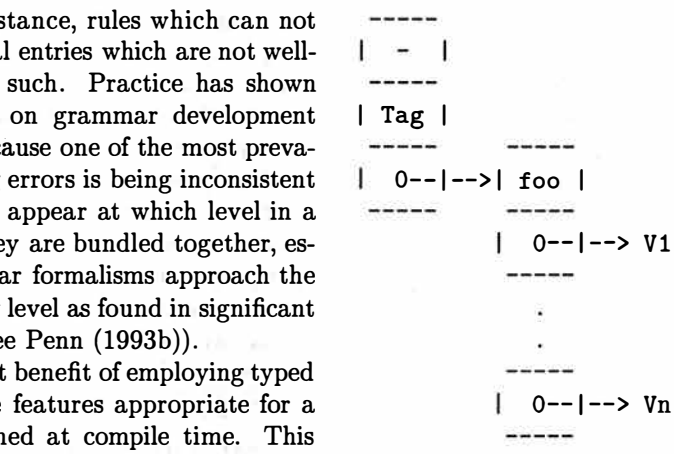
compile-time. For instance, rules which can not be satisfied and lexical entries which are not well-typed are flagged as such. Practice has shown that this cuts down on grammar development time significantly, because one of the most prevalent grammar-writing errors is being inconsistent about which features appear at which level in a structure and how they are bundled together, especially when grammar formalisms approach the 200 node lexical entry level as found in significant fragments of HPSG (see Penn (1993b)).

Another significant benefit of employing typed structures is that the features appropriate for a type can be determined at compile time. This has two advantages. First, it allows memory allocation and deallocation to be handled efficiently, as the type of each structure is known. Second, it allows unification to be greatly speeded up as there is no need to merge features represented as lists; the positions of relevant features are known at compile time. We consider these two benefits in turn.

ALE is currently implemented in Prolog, though plans are underway to implement it in C, using WAM techniques directly. As things stand, the WAM implementation of Prolog is exploited heavily to develop WAM-like behavior for ALE. Using Prolog for feature structure unification systems has its advantages and drawbacks. The drawback is that there are no pointers in Prolog, and thus path compression during dereferencing can not be carried out efficiently (though it is carried out on inactive edges during parsing). The advantage is that Prolog is very good at structure copying, last call optimization, incremental clause evaluation and search. We will consider all of these topics. But first, we note that the data structure used for feature structures in ALE is:

```
Tag-foo(V1,...,Vn)
```

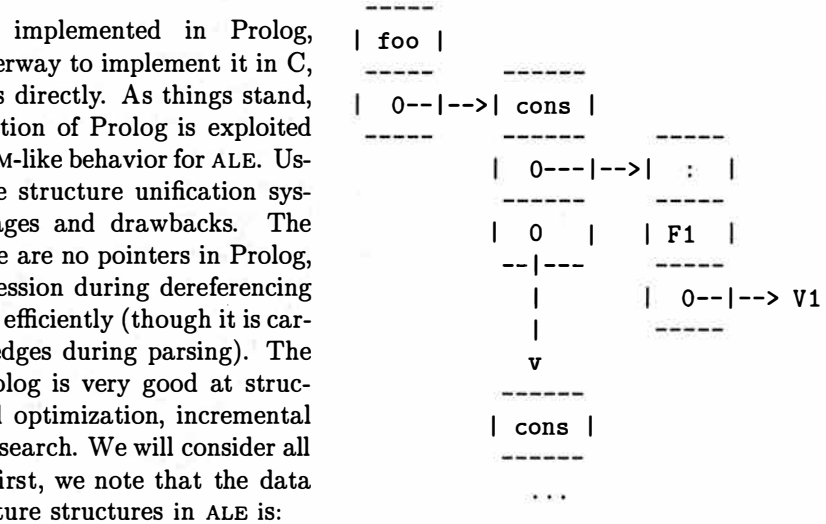
where `Tag` is a reference pointer, signalling the intensional identity of the structure, much as a position in memory in an imperative language would do, and where `foo` is the name of the type of the structure, which must be a Prolog atom, of course, and where `V1` through `Vn` are the values for the features `F1` through `Fn` that are appropriate for type `foo`. Given Prolog's compilation to the WAM, this amounts to having the following kind of record structure for feature structures:



Contrast this with a representation such as:

```
Tag-foo([F1:V1,...,Fn:Vn])
```

where the features are coded explicitly in terms of a list. Here the structure required is as follows (ignoring the tag):



In general, our representation requires $4+n$ cells for a structure with n features, while the usual one requires $4 + 6n$ cells for the same structure. This constitutes a huge discrepancy when we consider the amount of overhead this induces throughout the grammar in areas such as lexical retrieval and copying edges into the chart. Note that this difference between using record-like structures as opposed to lists of feature-value pairs is Prolog-independent.

We have not said much about the tag. It is based on the same principle as O'Keefe's method

of encoding arrays in Prolog using variables, which provides constant time access and update (see the Quintus library). The basic idea is that each slot in an array is associated with a value and a pointer, which is either a variable or a structure consisting of another value and a pointer. Updates are performed by instantiating the variable to a new pair consisting of a variable and value. Thus values are found by tracking the pointer until it's a variable. To maintain constant time, the entire array must be regularly updated. In our case, the tag plays the role of the pointer, and dereferencing is performed by following the tag value until it is a variable. The number of dereferencing steps needed at any stage is bounded by the depth of the inheritance hierarchy (of course, Prolog does its own internal dereferencing, so we can not statically bound the total number of dereferencing steps needed during unification). Path compression is then the equivalent of O'Keefe's array updating, and is performed when a completed edge is found during parsing. We will see examples of the use of tags shortly.

The second benefit we mentioned for typing structures is that we are able to carry out unification without merging feature-value lists. The standard method in unifying feature structures is to take two lists of features, find the common elements, unify them, and take both symmetric differences and copy the results of this into the final result. Such tasks are extremely costly, especially as the number of features grows. Instead, our compiler will produce the following kind of code to perform unification (which has been simplified here, but will be expanded upon later):

```
unify(T-ne_list(H,R),T-ne_list(H2,R2)):-
  unify(H,H2),  unify(R,R2).
```

The positional encoding of feature values means that at compile time, we know which features any two types have. Also note that the tags are identified to make sure any update of one structure is felt by the other. Compare the number of operations required in the above procedure to one that had to look through two lists of feature value pairs, and act conditionally depending on the results of comparisons, and detect termination conditions. While our unification in the above case requires only two logical operations in Prolog, the list merging method would require at least two comparisons, a termination test and the same two logical operations as our method, more than doubling the cost in the best possible instance. Of course, matters are much worse if the features are out of order or don't line up exactly in the two structures.

Having motivated our approach, we now consider two operations on feature structures which are calculated at compile time. The first of our operations on feature structures involves adding the information that it is of a given type. For instance, we might have a list and want to add the information that it is non-empty. In this case, the system produces the following code:

```
add_to(ne_list,Tag-TVs):-
  add_to_ne_list(TVs,Tag).

add_to_ne_list(list,
               _-ne_list(T1-bot,
                         T2-bot)):-
  add_to_atom(bot,T1),
  add_to_list(bot,T2).
```

Note that to add the information that a structure is a `ne_list` at the top level, we make the first argument the type-value term. The reason for doing this is that the WAM performs first-argument indexing. This means that given the current structure, of type `list`, hashing is done to find the code to add the fact that it is a non-empty list to it. In fact, the clauses for `add_to/2` are never used at run-time, as can be seen from the second of the above clauses, which call `add_to_atom(TVs,Tag1)` rather than `add_to(atom,Tag1-TVs)`. At this stage, we should point out that it would be more efficient to use the following code:

```
add_to_ne_list(list,
               _-ne_list(_-atom,_-list)).
```

For completely fresh features such as the head and tail above, there is really no reason to create a structure `Tag-bot` and then immediately add a type to it. The next release of ALE (Carpenter and Penn forthcoming) will have such an optimization, as it is statically computable.

On the other hand, consider the effect of adding the type `word` to the type `sign` given above:

```
add_to_word(sign(T1-Phon,SynSem,QSt),
            _-word(T-Phon,SynSem,QSt)):-
  add_to_singleton_phon_list(Phon,T).
```

Here we see that (pointers to) the feature values for `sign` are copied over into the new word structure created and the additional constraint that the `Phon value` be a singleton list must also be resolved. Note that this extra bit of (pointer) copying is something that is usually also done in encodings using feature-value pairs.

As we hinted at above, the procedure to perform unification on two structures is also compiled before run-time. In particular, consider the code to unify two `ne_lists`, in its full form:

```
unify_deref(FS1,FS2):-
  deref(FS1,Tag1,TVs1),
  deref(FS2,Tag2,TVs2),
  ( Tag1 == Tag2, !
  ; unify(TVs1,TVs2,Tag1,Tag2)
  ).

unify(ne_list(H1,T1),TVs2,Tag1,Tag2):-
  unify_ne_list(TVs2,H1,T1,Tag1,Tag2).

unify_ne_list(ne_list(H2,T2),H1,T1,T,T):-
  unify_deref(H1,H2),
  unify_deref(T1,T2).
```

The strange argument order and extra level of indirection comes about to exploit the first-argument indexing of the WAM. In effect, what happens when unifying two structures is that they are first dereferenced, then two hashings are performed, one on each of their types, and finally their shared feature values are unified. This illustrates one of the simplest cases of unification. Note that absolutely no type inference is required at run time because the compiler knows that when two structures of the same type are unified, then their features already meet the type constraints, and hence so will the result of unifying them. Other cases might involve `add_to_`*sort*`/2` goals being called and tags being instantiated, when unifying the two structures leads to a new structure with a type higher than each of the inputs. For instance, suppose we have:

```
b sub [c] intro [f:x,h:u].
d sub [c] intro [g:y,h:v].
  c sub [] intro [f:x2,h:u+v,j:z].
```

where `u+v` is the type unification of `u` and `v`. Then we would have:

```
unify(b(V1,V2),TVs2,Tag1,Tag2):-
  unify_b(TVs2,V1,V2,Tag1,Tag2).

unify_b(d(V3,V4),V1,V2,
        _-c(V1,V2,V3,T-bot),
        _-c(V1,V2,V3,T-bot)):-
  unify_deref(V2,V4),
  deref(V1,Tag1,SVs1),
  add_to_x2(SVs1,Tag1),
  add_to_z(bot,T).
```

When unifying structures of type `b` and `d`, we must instantiate both of their reference pointers to a new structure of type `c`, with a new feature `j`, and in addition, perform the extra type inference on the value of `f`. It is worth noting that all and only the necessary type inference is determined at compile-time. For instance, the fact that the `h` value of `c` is required to satisfy the unification of the constraints on `h` in `b` and `d` is enough to let the compiler determine that no additional type inference will be required.

3 Compiling Descriptions

In this section, we consider compiling descriptions taken from ALE's attribute-value logic:

```
<desc> ::= <type>
         | <var>
         | <feat>:<desc>
         | <desc> and <desc>
         | <desc> or <desc>
```

As was shown by Smolka (1988), the lack of variables can lead to a quadratic increase in the size of descriptions using only path equations; with variables, path equations are no longer necessary. A complete proof theory with respect to both an algebraic semantics and a feature-structure based interpretation can be found in (Carpenter 1992).

Descriptions are compiled into the operations of `add_to_`*sort*, `unify`, `deref`, and a combination of conjunction and disjunction in Prolog. In addition, to handle constraints of the form `<feat>:<desc>`, which tell us to add the description to the value of the feature, we need a procedure for extracting a feature's value from a structure. This is done with clauses such as:

```
featval(hd,FS,Val):-
  deref(FS,Tag,TVs),
  featval_hd(TVs,Tag,Val).
```

```
featval_hd(ne_list(H,_),Tag,H).
```

Again, we present the first clause for convenience; only the second is used at run-time, combined with the necessary dereferencing. Note that if we look for the `hd` value of a structure of type `list`, we coerce `list` to `ne_list`:

```
featval_hd(list,_-ne_list(T-atom,_-list),
                T-atom).
```

Here we create a new structure of type `ne_list`, with a fresh head and tail, and return the fresh head as the result. In general, this might require additional type inference, as could be seen by considering what would happen if we took the value of the feature `j` in an object of type `d` in the above type system. In this case, the type `d` object would be coerced to one of type `c`, which in turn requires boosting the type of its `h` value and adding new `f` and `j` values:

```
featval_j(d(V1,T2-TVs2),
          _-c(T3-bot,V1,T2-TVs2,T-bot),
          T-bot):-
  add_to_u(TVs2,T2),
  add_to_x2(bot,T3),
  add_to_z(bot,T).
```

Again notice that the compiler determines exactly which type inferences to perform as part of finding a feature's value. Again, in the next release of ALE, the `add_to_`*sort*`(bot,T)` goals will be replaced with instantiated feature structures of type *sort*.

We are now in a position to see how descriptions get compiled into Prolog clqauses. To add a description of the sort found on the left to a dereferenced structure `Tag-TVs`, the Prolog code on the right is generated:

```
sort          add_to_sort(TVs,Tag)

V             deref(V,Tag2,TVs2),
              unify(TVs1,TVs2,Tag1,Tag2)

f:D           featval_f(TVs,Tag,Val),
              deref(Val,Tag2,TVs2),
              [ add D to Tag2-TVs2 ]

D1 and D2     [ add D1 to Tag-TVs ],
              deref(Tag-TVs,Tag2,TVs2),
              [ add D2 to Tag2-TVs2 ]

D1 or D2      ( [ add D1 to Tag-TVs]
              ; [ add D2 to Tag-TVs]
              )
```

Sorts are straightforward, and simply invoke the appropriate `add_to` goal. Variables are such that they get instantiated to the feature structures which they describe. Thus adding a variable to a structure involves dereferencing the variable, which is instantiated to the current value it has, and unifying it with the structure to which it is being added. All variables are initialized to `Tag-bot` at compile-time for compatibility with the basic operations over feature structures. The last three cases are recursive. Adding a description to a feature's value requires finding the feature's value, dereferencing it, and adding the embedded description. Conjunction and disjunction in descriptions are translated into the corresponding Prolog control structures. In particular, this means that we treat disjunction in descriptions as introducing non-determinism in adding a description. In this way, Prolog backtracking, and its attendant efficient implementation of search and variables, will take care of the disjunction without any need for explicit copying in the program. Of course, it's still there — it's just that Prolog's doing it. In a non-Prolog implementation of this method, a programmer would have to be very clever to implement this kind of control structure, using some kind of lazy copying along the lines of Tomabechi (1992) or along the lines of the WAM itself. Conjunction, on the other hand, is treated as goal sequencing in Prolog.

4 Compiling Grammars and Programs

This compilation of descriptions into Prolog code rather than into feature structures is where ALE departs most radically from other attribute-value-based parsers of which we are familiar. The traditional method, say for chart parsing, involves taking an inactive edge which has just been created and trying to unify it with the feature structures corresponding to the heads of rules in the grammar. Instead, our system will execute the Prolog code compiled from the description of the

head of a grammar rule. There are two principal benefits to our approach. These stem from the fact that we reduce the copying and search methods to those of the WAM itself by compiling the Prolog clauses generated. The first benefit is that early failures in matching a description to a goal do not result in any overcopying — in fact there is really no copying done at all — it's all handled in the heap mechanism of the WAM. The second benefit is that if we have deeply embedded disjunctions in our descriptions, we do not need to expand to a disjunctive normal form or invoke one of the many approaches to disjunctive unification. In particular, if we have a description with an embedded disjunction and the first disjunct fails, then we only backtrack to the second disjunct, not all the way back to the beginning of the structure. Again, this operation is very efficient in the WAM. It should be noted that nothing here depends on using a chart parser as the control strategy — similar benefits would accrue to any other parsing strategy. In fact, the same benefits could also be gained by using this kind of strategy in generation, say along the lines of van Noord et al. (1992).

The chart parsing strategy used in ALE is not particularly significant qua parser, as it was primarily motivated by Prolog considerations. What is significant is the way in which descriptions are compiled and made available to the parser, a strategy which can be maintained using many different parsers. For instance, we are also working on a left-corner parser which will not require any copying or manipulation of the database. The most significant thing to note about ALE's parser is that it employs a dynamic chart, where inactive edges are asserted into the database,[1] and parses according to a bottom-up strategy, from right to left in the chart, and from left to right through individual rules. Active edges are truly active, being represented only by the current position in a Prolog clause compiled from a rule description. Rules are of the form:

```
D0 ===> D1, ..., DN.
```

where D0 is the description of the mother category and the Di are descriptions of the daughter categories. The grammar rules are then compiled into the `rule/3` predicate. The goal `rule(L,R,C)` is called whenever an inactive edge `C` is added from position `L` to `R` in the chart, either by the lexicon or by `rule/3` itself. The code produced for the above rule is:

```
rule(C1,Left,Mid1):-
  [ add D1 to C1 ],
  edge(Mid1,Mid2,C2),
  [ add D2 to C2 ],
  ...
  edge(MidN-1,Right,CN),
  [ add DN to CN ],
  [ add D0 to Tag-bot ],
  fully_deref(Tag-bot,C0),
  assert(edge(Left,Right,C0)),
  rule(Left,Right,C0).
```

When `rule(C1,Left,Mid1)` is called, the first thing that happens is the description `D1` being added to the feature structure `C1`. Assuming this fails, no other work is done, and no copying is performed. Instead, the code generated by the description `D1` is simply executed, and failure causes Prolog backtracking either to earlier disjunctions in the description `D1`, or to other clauses for `rule/3` generated by other rules. Assuming `D1` is successfully added to `C1`, `rule/3` looks for an inactive edge directly to the right of `C1` in the chart. The fact that parsing is done right to left ensures that the chart has been completed to the right of any inactive edge which is being considered. If an inactive edge of category `C2` to the right is found, `rule/3` attempts to add the description `D2` to a copy of `C2`. The current bottleneck in this process is the inordinate amount of copying required, especially when many empty categories are present. A better solution would be to add the descriptions in a more lazy fashion without eagerly copying the whole structure, but Prolog does not provide that kind of fine control of its database. This process continues until the right hand side of the rule is completely matched. At this point, the mother category is constructed by adding the compiled description `D0` to a fresh category, fully dereferencing (path compressing), asserting it into the database of inactive edges, and recursively calling `rule/3`. As there are no

[1] Current versions of the WAM in SICStus and Quintus index asserted clauses, allowing the edges beginning at a particular position to be easily retrieved by hashing — a method with explicit copying would most likely be faster than the one with `assert`, and we plan to explore this possibility.

base cases to `rule`, it will eventually fail and backtrack through all of the disjunctive choice points and alternative rules.

The input string is consumed from right to left, at each step adding inactive edges until no more edges can be added. This gives the parser as a whole a mix of breadth-first and depth-first search, to best exploit the inherent behavior of the WAM. The top level control strategy is quite straightforward:

```
parse(Words,C):-
  reverse(Words,WordsRev),
  length(Words,N),
  build_chart(Words,N),
  edge(0,N,C).

build_chart(_,N):-
  empty(C),
  assert(edge(N,N,C)),
  rule(N,N,C).
build_chart([],_).
build_chart([W|Ws],N):-
  M is N-1,
  ( lex(W,C),
    assert(edge(M,N,C)),
    rule(M,N,C)
  ; build_chart(Ws,M)
  ).
```

The words are reversed and counted, and the chart is built from the right to left, taking lexical entries for each word and firing `rule/3`. Before considering lexical entries, empty categories are asserted into the chart and processed using `rule/3`. All lexical and empty category alternatives will be considered during backtracking before proceeding leftward to the next word. We should also mention that lexical entries and empty categories are fully expanded as path-compressed feature structures at compile time.

In addition to allowing categories in a rule, ALE also allows definite clause goals to be invoked, in a way similar to DCG rules such as:

```
f(Z) ---> h(Y), g(X), {foo(X,Y,Z)}, j(Z).
```

In this rule, as soon as the `h(Y)` and `g(X)` categories are found, the goal `foo(X,Y,Z)` is invoked and solved before going on to consider `j(Z)`. The change to `rule/3` is minimal; the code for solving `foo(X,Y,Z)` is simply inserted in between the code generated by the categories `g(X)` and `j(Z)`.

Definite clause programs can be defined in ALE, where instead of Prolog terms, feature structure descriptions are used. For instance, we can define standard predicates such as:

```
append(e_list,X,X) if true.

append(hd:X and tl:Xs,
       Ys,
       hd:X and tl:Zs) if
  append(Xs,Ys,Zs).
```

The logical variables are used as in Prolog, with the result being an instance of constraint logic programming over the typed feature structure logic. This bears a close similarity to the LOGIN language of Aït-Kaci — Nasr (1986), who point out a number of benefits of using an order-sorted notion of feature structure for logic programming. A general CLP scheme suiting this application was defined by Höhfeld — Smolka (1988) and this particular application is detailed in (Carpenter 1992). The previous two clauses will translate into the following pieces of code, following O'Keefe's (1990) meta-interpreters (and omitting all of the dereferencing):

```
solve([]).
solve([G|Gs]):-
  solve(G,Gs).

solve(append(FS1,FS2,FS3),Goals):-
  add_to_e_list(TVs1,Tag1),
  unify(FS2,FS3),
  solve(Goals).
solve(append(FS1,FS2,FS3),Goals):-
  featval_hd(TVs1,Tag1,FS1H),
  featval_hd(TVs3,Tag3,FS3H),
  unify(FS1H,FS3H),
  featval_tl(TVs1,Tag1,FS1T),
  featval_tl(TVs3,Tag3,FS3T),
  solve(append(FS1T,FS2,FS3T),Goals).
```

The coding used, with goals being threaded, is to ensure that last call optimization takes place. While ALE does not perform indexing, it does support full cuts, disjunctions, negations by failure and last call optimization.

Such procedural attachments can be interspersed into rules just as in DCGs. This mechanism has been used in ALE grammars for purposes such as quantifier scoping using Cooper

Storage, for treating the maximal onset principal in syllabification in attribute-value phonology (Mastroianni 1993), and for implementing principles such as the non-local feature principle (for slashes) and the binding theory of HPSG (Penn 1993b). Procedures can even be used to postpone some of the unifications in a rule until after all of the categories have been found, thus encoding a form of restriction similar to that used by Shieber (1985). Such procedures will allow general hooks to Prolog in the next release of ALE, and as the definite clause component of a grammar can be arbitrary, can also be used for interleaving on-line semantic processing with syntactic processing as in Pereira — Pollack (1990).

Before concluding, we should also point out that ALE has a number of other useful features. One of the most interesting of these is the use of lexical rules, which are loosely based on those of PATR-II, in that they map one lexical entry to another at compile-time. In ALE, such rules may involve procedural attachments just as other rules, and contain a rudimentary morphological component based on string unification. ALE also fully supports parametric macros which are compiled out statically into the descriptions they abbreviate.

The next release of ALE, scheduled for Summer 1993, will also include more general constraints on types, following Aït-Kaci (1986) (see also Carpenter (1992)), inequations and extensionality (see Carpenter (1992) for theoretical details, and Penn — Carpenter (forthcoming) and Penn (1993a) for implementation details and motivation).

5 Conclusion

We have shown how grammars based on attribute-value logic descriptions can be efficiently compiled into low-level Prolog instructions which exploit the inherent efficiency of the WAM. Unfortunately, there are a few inefficiencies stemming from this encoding due to Prolog's logical variables and its lack of control over copying structures from the database. The ideal solution will be to build a WAM-like abstract machine language directly for typed feature structures and their associated descriptions. The WAM has proved to be the most efficient architecture yet developed for implementing "unification-based" programs, even though, as we saw, it often relies on structure copying and creation rather than unification (the only cases of unification in ALE arise from shared variables in a structure — everything else is structure copying).

Current benchmarks, using the standard naive reverse, with just the definite clause component of ALE, place it at roughly 1000 logical inferences per second (LI/s) on a DEC 5100 running SICStus 2.1, which is roughly 1.5% of the speed of the SICStus compiler itself. HPSG grammars where lexical entries run between 100 and 200 words, all of the local principles have been implemented according to Pollard and Sag (in press), process 15 word sentences, creating 40-50 inactive edges, at times under 2 seconds.

ALE Version β, as described in this paper, is available from the author without charge for research purposes. It runs under SICStus and Quintus Prologs. It is distributed with roughly 100 pages of documentation and sample grammars. Version 1.0 is scheduled for release in August 1993.

References

Aït-Kaci, H. (1986a). An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45.

Aït-Kaci, Hassan — Roger Nasr (1986) LOGIN: A logic programming language with built-in inheritance. Journal of Logic Programming 3.

Aït-Kaci, Hassan (1991) *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press.

Carpenter, Bob (1992) *The Logic of Typed Feature Structures.* Cambridge University Press.

Carpenter, Bob (1993) ALE User's Guide — β. Laboratory for Computational Linguistics Technical Report, Carnegie Mellon University, Pittsburgh.

Carpenter, Bob — Gerald Penn (forthcoming) ALE User's Guide Version 1.0. Laboratory for Computational Linguistics Technical Report, Carnegie Mellon University, Pittsburgh.

Carpenter, Bob — Carl Pollard (1991) Inclusion, disjointness and choice: the logic of linguistic classification. *Proceedings of the ACL.*

Höhfeld, M. — Gert Smolka (1988) Definite relations over constraint languages. LILOG Report 53, IBM, Stuttgart.

Kasper, Bob — Bill Rounds (1990) The logic of unification in grammar. Linguistics and Philosophy 13.

Mastroianni, Michael (1993) Attribute-logic Phonology. MS Thesis, Computational Linguistics Program, Carnegie Mellon University, Pittsburgh.

Meseguer, J. — J. Goguen — G. Smolka (1987) Order-sorted unification. CSLI Report 87-86, Stanford.

O'Keefe, Richard (1990) *The Craft of Prolog.* MIT Press.

Penn, Gerald (1993a) A utility for typed feature structure-based grammatical theories. MS Thesis. Computational Linguistics Program, Carnegie Mellon University, Pittsburgh.

Penn, Gerald (1993b) A comprehensive HPSG grammar in ALE. Laboratory for Computatonal Linguistics Technical Report. Carnegie Mellon University, Pittsburgh.

Pereira, Fernando — David H. D. Warren (1980) Definite clause grammars for language analysis. *Artificial Intelligence* 13.

Pereira, Fernando — Martha Pollock (1991) Incremental interpretation. *Artificial Intelligence* 50.

Pollard, Carl — Ivan Sag (in press) *Head-Driven Phrase Structure Grammar.* CSLI/University of Chicago Press.

Shieber, Stuart (1985) Using restriction to extend parsing algorithms for complex-feature-based formalisms. Proceedings of the ACL.

Smolka, Gert (1988) A feature logic with subsorts. LILOG Report 55. IBM, Stuttgart.

Tomabechi, Hideto (1992) *Quasi-Destructive Unification.* PhD Thesis, Computational Linguistics Program, Carnegie Mellon University, Pittsburgh.

(Pictorial) LR Parsing from an Arbitrary Starting Point

Gennaro Costagliola

Dipartimento di Informatica ed Applicazioni,University of Salerno
I-84081 Baronissi, Salerno, Italy
email: gencos@udsab.dia.unisa.it

Abstract

In pictorial LR parsing it is always difficult to establish from which point of a picture the parsing process has to start. This paper introduces an algorithm that allows any element of the input to be considered as the starting one and, at the same time, assures that the parsing process is not compromised. The algorithm is first described on string grammars seen as a subclass of pictorial grammars and then adapted to the two-dimensional case. The extensions to generalized LR parsing and pictorial generalized LR parsing are immediate.

1 Introduction

This paper introduces an LR algorithm for the parsing of input whose starting point is not defined. The main motivation behind this comes from the area of the two-dimensional LR parsing. Given a two-dimensional pattern it is not always obvious how to determine the starting point from which the parsing process should begin. The proposed algorithm avoids this problem allowing any element of the pattern to be considered as the starting one and, at the same time, it assures that the parsing process is not compromised.

The main idea is to create two substring LR parsers, one for the given language and the other for the "reverse" version of the language. The two parsers proceed in parallel, scanning the input in opposite directions, from the designated starting element. Neither of the two is allowed to reduce beyond their parser stack. When this is required, a rendezvous with the other parser must occur and both must perform the same reduction. The two parser stacks can be considered as an only graph stack expanding to the right and to the left of a starting point.

This paper describes the algorithm on string grammars and then shows an extension to the case of positional (two-dimensional) grammars. The algorithm can be easily extended to treat languages whose LR parsing tables present conflicts. In fact, the use of the graph stack is the same as the one adopted in Tomita's generalized parser.

Section 2 contains comments on the work related to this paper; in Section 3, the preliminary definitions of *reverse grammar* and of *joint graph* are given; Section 4 presents the data structures and the description of the algorithm; in Section 5 the algorithm is adapted to the two-dimensional LR parsing after the description of the positional grammars and positional parsing tables. Section 5 contains the Conclusions.

2 Related work

This section contains two parts: one relates to the pictorial parsing that is the main motivation behind this paper and the other relates to island-driven parsing, since the algorithm presented in this paper can be considered as a bidirectional LR parser.

2.1 Pictorial parsing

With the introduction of more and more powerful graphical interfaces, the interest in the study of pictorial parsing is increasing. At the moment, many parsers have been designed, each of them having advantages and disadvantages with rispect to one another.

A recently proposed classification, (Wittenburg, 1992), considers two major classes: bottom-up order-free pictorial parsers (Crimi *et al.*, 1991; Golin, 1991; Helm, 1991; Wittenburg, 1991) and predictive pictorial parsers (Costagliola — Chang, 1991; Wittenburg, 1991).

The main advantage of an order-free parser over a predictive one is that it can compose the input objects in any order and it is not bound to a mandatory pre-ordered navigation of the input. This gives great expressive power to the underlying grammar formalisms.

The input data structure for an order-free parser is made by two sets: a set of objects and the set of all the relations among them. The relations must be the same used in the parser. The parser then proceeds with a purely bottom-up enumeration.

The predictive pictorial parsers direct the order in which the objects in the input space are processed by the parser. This limits the expressive power of the underlying grammar formalisms but still retain expressive power enough to describe many interesting 2D languages like arithmetic expressions, lines, document layouts, some class of graphs, etc.

The input data structure only contains the set of the objects with their attributes and does not need to keep information about the relations among the objects. The relations are embedded in the parser that use them to predict the attribute values of the next object to parse. This representation is more space efficient than the other and does not depend on the particular parser relations. Moreover, it refers to a relation only when necessary.

The predictive nature of the parser makes it more efficient than an order-free bottom-up pictorial parser. In particular, for pictorial LR parsing it is even possible to use tools from string-based formal language theory like Yacc for the automatic parser generation of a pictorial language (Costagliola *et al.*, 1993).

However the prediction of the next object induces an order on the visit of the input. The order can be linear (Costagliola — Chang 1991) or partial (Wittenburg, 1992) and, in any case, it forces the parser to begin its processing from one (linear case) or multiple (partial case) specific starting points. If the input is made of a set of objects with no indication on the starting point, like a scanned document layout, then predictive parsing becomes inefficient. This paper attempts to solve this problem by constructing a bidirectional LR parser that does not need specific starting points in the input.

2.2 Island-driven parsing

Island-driven parsers are generally used for generating partial interpretations of a spoken sentence (Stock *et al.*, 1989; Woods, 1982). The parsing starts from words that have higher acoustic evidence and then extends to both directions in the sentence. Each partial interpretation forms an "island". Occasionally, two islands may 'collide' by proposing and discovering the same word in the gap among them and may then be combined into a single larger island.

This can be effectively used in pictorial parsing whenever there are objects of particular semantic relevance, or objects particularly complex to be combined only when each of them has been recognized, or, in our case, the starting point is not easy to find.

Other approaches to bidirectional parsing include bidirectional chart parsing (Stock *et al.*, 1989; Steel — De Roeck, 1987) and some form of bidirectionality within a tabular approach, such as Earley's or Kasami-Cocke-Younger's (Bossi *et al.*, 1983).

3 Some definitions

This section contains two definitions that will be useful for the presentation of the final algorithm.

Definition 1 (reverse grammar) *Given a context-free grammar $G = (N, T, S, P)$, a reverse grammar with respect to G is a new grammar $G' = (N, T, S, P')$, where P' is defined as follows: whenever $A := u$ is in P then $A := u^R$ is in P', where u^R is the reverse version of u.*

In general, the reverse of an LR context-free grammar is not LR.

For sake of simplicity, this paper considers only LR context-free grammars whose reverse is LR, too. The extension to general context free-grammars can be easily done.

Example 3.1.

The grammar G:
(1) S := CC
(2) C := cC
(3) C := d

The reverse grammar G′:
(1) S := CC
(2) C := Cc
(3) C := d

It is assumed that corresponding productions have the same ordering number. Note that if the C's in the production S := CC are indexed, then the grammar G contains the production S := C_1C_2 and the grammar G′ contains S := C_2C_1. To formalize this concept, let us index all the occurrences of symbols on the right side of the productions of G such that an into correspondence between occurrences of symbols and indices is created. After indexing, the grammar G becomes:

(1) S := C_1C_2
(2) C := c_3C_4
(3) C := d_5

C_1, an occurrence of C, is now different from the occurrence C_2 but

$$\text{name}(C_1) = \text{name}(C_2) = C.$$

Given an LR grammar G such that its reverse grammar G′ is LR, it is always possible to construct for each of them the canonical LR(0) collection of sets of items through the algorithms *Closure*, *Goto* and *Set-of-Items Construction* as defined in (Aho *et al.*,1985). The goto graphs for G and G′ are shown in Figure 3.1.

Let us define $C_I = I_0, \ldots, I_n$ the collection of sets of items for G and $C_R = R_0, \ldots, R_m$ the collection of sets of items for G′. In the following, the relation between elements of C_I and C_R is analyzed. It is assumed that no useless symbols or epsilon-productions are in G.

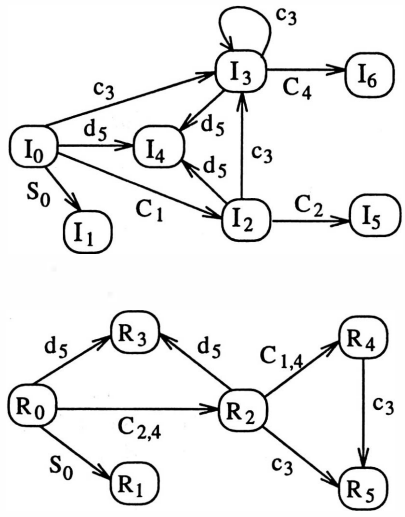


Figure 3.1 Goto Graphs for G and G′

Given a production "A := u X_i v"' in G with u, v ∈ (N ∪ T)* and X_i ∈ (N ∪ T), there must be a set-of-items I_k reachable after the occurrence X_i has been processed, i.e., a set-of-items I_k containing the item "A := u X_i . v".

If the corresponding production in G′ "A := v^R X_i u^R" is considered, there must exists a set-of-items R_l reachable after the occurrence X_i has been processed, i.e., a set-of-items R_l containing the item "A := v^R X_i . u^R". Here v^R and u^R are again the reverse versions of v and u, respectively

In other words, if I_k is the state reachable after a forward scanning of X_i in the context of u X_i v, then there must exists R_l, the state reachable after a backward scanning of X_i still in the context of u X_i v.

As an example, the goto graphs above show that I_3 and R_5 are both reachable through c_3.

Definition 2 (joint graph) *Let us consider a grammar G = (N, T, S, P) with indexing, its reverse grammar G′ = (N, T, S, P′) and their canonical LR(0) sets-of-items collections* $C_I = I_0, \ldots, I_n$ *and* $C_R = R_0, \ldots, R_m$, *respectively.*

The joint graph *for an* occurrence X_i *of* X *in* $N \cup T$ *is given by:*

Jgraph(X_i) = { (I_k, R_l) / *there exist* $A \in N$, *and u, v occurrences of strings* $\in (N \cup T)^*$ *such that "*$A := u\ X_i\ .\ v$*"* $\in I_k$ *AND "*$A := v^R\ X_i\ .\ u^R$*"* $\in R_l$ }

The joint graph *for the* symbol X *in* $N \cup T$ *is:*

$$Jgraph(X) = \bigcup_{i:name(X_i)=X} Jgraph(X_i)$$

Example 3.2.

Considering the grammars in Example 3.1 and looking at the goto graphs above:

Jgraph(c) = Jgraph(c_3) = {(I_3, R_5) }
Jgraph(d) = Jgraph(d_5) = {(I_4, R_3) }
Jgraph(S) = Jgraph(S_0) = {(I_1, R_1) }
Jgraph(C) = Jgraph(C_1) ∪ Jgraph(C_2) ∪
∪ Jgraph(C_4) = {(I_2, R_4) } ∪ {(I_5, R_2) } ∪
∪ {(I_6, R_2), (I_6, R_4) } =
= {(I_2, R_4), (I_5, R_2), (I_6, R_2), (I_6, R_4) }

To parse the string "d′cd" starting from the symbol "c", a substring parser based on G will parse "cd" and a substring parser based on G′ will parse "cd′".

As Jgraph(c) = {(I_3, R_5) }, the first parser will start from state I_3, while the second one will start from R_5. To parse the whole string the completion of the substring parser based on G will have to match the parsed part of the reverse substring parser, and vice versa.

After reducing the non-terminal C, the states of G to be considered are I_2, I_5 and I_6 while the states of G′ are R_2 and R_4. Note that if the parsing process starting in R_4 fails, then that starting in I_2 must fail, too, because it has no corresponding state R_i left in Jgraph(C).

4 The Algorithm

The algorithm is based on the concepts of substring parsing as presented in (Rekers — Koorn, 1991). In this paper, an algorithm for substring parsing for arbitrary context-free grammars is presented. It is based on the pseudo-parallel parsing algorithm of Tomita (Tomita, 1985, 1991), which runs a dynamically varying number of LR parsers in parallel and accepts general context-free grammars.

Even though the algorithm can be easily extended to the general case, in this paper it will be limited to accept only LR context-free grammars whose reverse is still an LR grammar. Informally the algorithm can be described as follow.

The input is given by a grammar G and its reverse grammar G′, the Jgraph for each symbol of the grammars, the two parsing tables and an input sentence $a_0 \ldots a_n$ with an index $0 \leq i \leq n$ from where the parsing process is supposed to start. Jgraph(a_i) provides the initial states. As seen before, a Jgraph contains states (set-of-items) from both the grammar G and its reverse.

In the following, a *forward parser* is an LR parser for G and a *backward parser* is an LR parser for G′. Moreover, the *opposite* parser of a forward parser is meant to be a backward parser and vice versa.

Every state I_k in Jgraph(a_i) (with i being the starting position) becomes the initial state for a forward parser and every state R_l in Jgraph(a_i) an initial state for a backward parser. The forward parsers interact only with forward parsers in the same way as a generalized LR parser. The same is true for the backward parsers. The exception occurs when a parser tries to reduce a production "A := u" requiring the stack to pop its initial state. That parser is then blocked waiting for an opposite parser to try the corresponding reduction "A := u^R" on the same symbols.

If the distance between the two parser stack tops is | u | and the initial states of the two parsers form an edge in Jgraph(a_i), then a *rendezvous* occurs and Jgraph(A) is generated. Jgraph(A) will produce a new set of forward and backward parsers and the process will continue till when either two opposite parsers have a rendezvous on the action "accept" or no rendezvous is possible and the input has all been consumed.

4.1 Data Structures

The algorithm is based on a graph-structured stack with two types of nodes: *joint* stack nodes and *simple* stack nodes and it is able to construct a packed shared parse forest, (Tomita, 1985, 1991).

A *joint stack node* is a 5-tuple (Jgraph, X, blast, flast, sf_ptr) where Jgraph is as defined above; X is either a terminal or a non-terminal; blast and flast point to the last elements visited during the backward and forward parsing of the input, respectively; sf_ptr is a pointer to a node labeled X in the packed shared parse forest.

A *simple stack node* is a 4-tuple (state, X, last, sf_ptr) where state is the state reached by the parser and corresponds to a set of items; last is the last terminal parsed; X and sf_ptr are as above.

Note that a joint node ({..., (I, R), ...}, X, blast,flast, sf_ptr) represents a graph whose elements are simple stack nodes of the type (I, X, flast, sf_ptr) and (R, X, blast, sf_ptr) and the edges are defined in the Jgraph component.

The operations on the graph stack are the Splitting, Combining and Local Ambiguity Packing operations, (Tomita, 1985, 1991), as used in the definition of the Generalized LR Parser. The only difference regards the updating of the node fields flast and blast. The following are the definitions of two new operations that must be added to the previous.

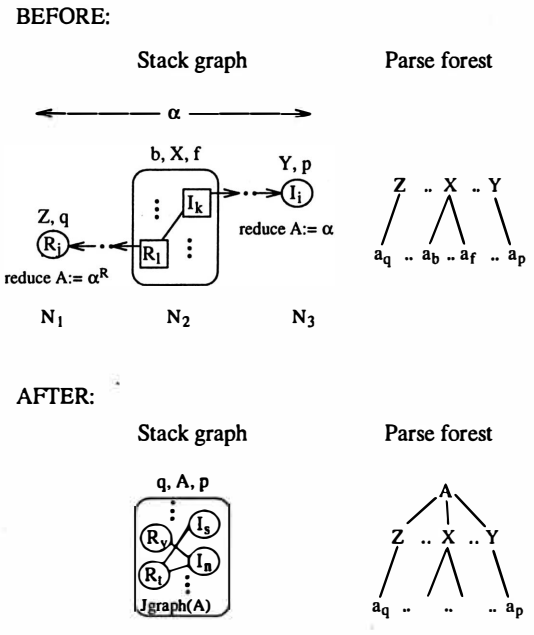


Figure 4.1 The Rendezvous Operation

The Rendezvous Operation

A graphical description of the rendezvous operation is given in Figure 4.1 (pointers from the stack graph to the parse forest are not shown).

If there is a joint node:
N_2 = ({ ... (I_k, R_l) ...}, X, b, f, X_ptr)
and two simple nodes:
N_1 = (R_j, Z, q, Z_ptr) and
N_3 = (I_i, Y, p, Y_ptr) such that

- N_1 is the active stack top of a backward parser with initial state R_l in N_2
- N_3 is the active stack top of a forward parser with initial state I_k in N_2
- the edge (I_k, R_l) is in the Jgraph of the node N_2.
- action(I_i, a_{p+1}) = "reduce A:= α" where α = Z ... X ... Y
- action(R_j, a_{q-1}) = "reduce A:= α^R"
- path_length(N_2 ... N_1) + path_length(N_2 ... N_3) - 1 = | α |

then N_1 and N_3 are made non-active and a new active joint stack node (Jgraph(A), A, q, p, A_ptr) is created, where A is the left-hand of the reduced production, q is the pointer to the last visited token in N_1, p is the pointer to the last visited token in N_3 and A_ptr is the pointer to a new shared forest vertex whose children are vertices pointed by stack nodes contained in the path N_1 ... N_2 ...N_3.

Note that the path N_2 ... N_1 represents the stack nodes of the backward parser while N_2 ... N_3 are the stack nodes for the forward parser.

The Accept Operation

If there is a joint node: N = ({ ... (I_k, R_l) ... S, 0, n, S_ptr)
where S is the starting non-terminal, 0 and n are the positions of the first and last elements of the input, and

- action(I_k, $) = accept
- action(R_l, $) = accept

then accept the input and return the pointer to the parse forest, S_ptr.

4.2 The LR parser with an arbitrary starting point

Input: An LR grammar G = (N, T, S, P) and its reverse G', the Jgraph for every symbol in N ∪ T, a sentence w = $a_0 a_1 \ldots a_n$ and a starting position i, with i ∈ {0, .., n}.
Output: The parse forest for w if accepted.
Method:

1. Create the LR parsing tables for G and G'

2. Create the joint node (Jgraph(a_i), a_i, i, i, a_i_ptr) and make it active.

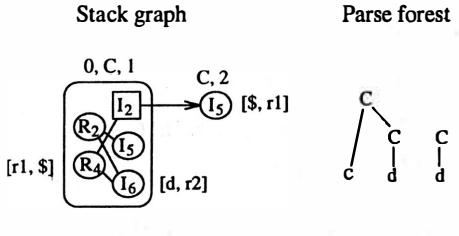
3. For every element I in the Jgraph field of an active joint node N = (Jgraph(X), X, b, f, X_ptr) start a forward generalized LR parser with initial node (I, X, f, X_ptr) on input $a_{f+1} \dots a_n$; for every element R in the Jgraph field of N start a backward generalized LR parser with initial node (R, X, b, X_ptr) on input $a_{b-1} \dots a_0$. Whenever a parser has a reduce action involving its initial node, or has an accept action to perform, make it wait. All the others will keep processing the input.

4. When all the parsers, both forward and backward, are in the wait state, apply the rendezvous operation wherever possible and go to step 3. If an accept operation is possible then return the corresponding pointer to the parse forest. If no rendezvous or accept operations are possible then the parsing process halts.

If no parse forest pointer has been returned then the sentence has not been accepted.

Example 4.1

Given the grammars G and G′ of Example 3.1, the input string "cdd" with positions 0, 1 and 2 respectively and the starting index 1, after two rendezvous operations the following situation is presented:

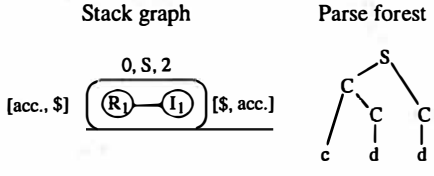


with the joint node ({(I_2, R_4), (I_5, R_2), (I_6, R_2), (I_6, R_4)}, C, 0, 1, C_ptr), the simple node (I_5, C, 2, C_ptr) and the corresponding [lookahead, action] pairs. The two nodes point to the first and the second tree in the parse forest, respectively.

On lookahead symbol \$, the state I_5 in the simple node of the forward parser built on G, requires a reduce action with production "(1) S := CC" while I_6, on lookahead d in position 2, requires a reduction with "(2) C:= cC". On the lookahead symbol \$, the state R_4 of the backward parser built on G′, requires the reduction "(1) S := CC". R_2 and I_5 have no action on \$ and 'd', respectively.

At this point, no action is possible without involving the joint node. Note that the nodes with states I_5 and R_4 meet the rendezvous operation requirements: both of them are active stack tops requiring the same reduce action, the sum of the depths of the stack of the two parsers - 1 = | CC | = 2 and (I_2, R_4) is an edge in the Jgraph of the joint node.

By applying the rendezvous operation on production "S := CC" and recalling that Jgraph(S) = {(I_1, R_1)} the following configuration is reached:



with the joint node ({(I_1, R_1)}, S, 0, 2, S_ptr) and I_1 and R_1 requiring both an accept action. The execution of the accept operation will then return the pointer to the final parse.

5 Two-dimensional LR parsing from an arbitrary starting point

An LR parser takes in input a sequence of tokens and returns a parse tree if the sequence is in the language accepted by the parser. The sequence of tokens are usually extracted from the string data structure.

A first generalization of this model toward 2-D parsing regards the possibility to have other input data structures different from the string. After all a string can be seen as a set of elements each having an attribute whose value is given by the position of the element in the string. As an example, the string "a b c" can be seen as the set {(b 2), (a 1), (c 3)} where each pair represents the element and its attribute value.

With this new data structure, the LR parser cannot simply require a "next" token to the lexical analyzer but has also to give indications on the position of the token. When this is done, the

input sequence of tokens to an LR parser can be extracted from any set of tokens with attributes. In the case of two-dimensional symbolic languages these attributes will correspond to Cartesian coordinates but other types of attributes can be thought of. In the case of diagrammatic languages, for example, size, shape, colour, etc. can be considered as attributes.

But how is it possible to make an LR parser give indications on the attribute values of the next token to parse? This can only be done by inserting appropriate information in the productions of the grammar from which the LR parser is built.

In the case of 2-D symbolic parsing this information is given by spatial operators that take in input the position of the last visited symbol and return the position of the next symbols to parse. Examples of spatial operators are:

"String Concatenation" : i ⇒ i + 1
"Up": (i, j) ⇒ (i, j+1)
"Left": (i, j) ⇒ (i-1, j)
"Right": (i, j) ⇒ (i+1, j)

5.1 Positional grammars

While in the traditional case there is an implicit use of the only string concatenation spatial relation, in the 2-D case many other spatial relations can be used and must made explicit in the grammar formalism.

In the following, some definitions are re-called, (Costagliola — Chang, 1991), to define a 2-D grammar formalism and the languages generated by it:

Definition 3 (positional grammar) *A context-free* positional grammar *PG is a six-tuple (N, T, S, P, POS, PE) where:*

N is a finite non-empty set of non-terminal symbols
T is a finite non-empty set of terminal symbols
$N \cap T = \phi$
$S \in N$ *is the starting symbol*
P is a finite set of productions
POS is a finite set of spatial relation identifiers
PE is a positional evaluator

Each production in P has the following form:

$$A := X_1\ Rel_1\ X_2\ Rel_2\ \ldots\ X_{m-1}\ Rel_m$$

where $m \geq 1$, $A \in N$, *each* X_i *is in* $N \cup T$ *and each* Rel_i *is in POS.*

In the following, the words "positional grammar" will also refer to a context-free positional grammar.

Definition 4 (pictorial language) *Let PG = (N, T, S, P, POS, PE). A* positional sentential form *is a string* II *such that* $S \Rightarrow^*$ II, *where* $\Rightarrow^*$ *has the conventional meaning. A* positional sentence *is a positional sentential form not containing non-terminal symbols. A* picture *is the evaluation of a positional sentence. The* pictorial language defined by a positional grammar L(PG) *is the set of its pictures.*

Note that if POS contains the only "string concatenation" spatial relation the positional grammar formalism reduces to the traditional context-free grammar formalism.

Example 5.1

The following positional grammar generates a simple subset of the arithmetic expressions:

N = {E, S, T, F}
T = {+, $\sum$, (,), id, num}
E is the starting symbol
POS = { >, _ }
P = { E := E > + > T | T
T := S > T | F
S := $\sum$ _ id
F := id _ id | num | (> E >) }

where the characters '>' and '_' stand for "horizontal concatenation" and "under concatenation", respectively. A positional sentence is:

"5 > + > $\sum$ _ i > (> x _ i > + > y _ i >)"

From its evaluation the particular positional evaluator PE for this grammar produces the following picture: $5 + \sum_i (x_i + y_i)$.

A more detailed definition of PE for this type of grammars can be found in (Costagliola *et al.*, 1992).

Example 5.2.

The following positional grammar PG will be used in the following to illustrate the execution of the algorithm.

(1) S := A Down S
(2) S := A
(3) A:= a Right a

A positional sentential form for this grammar is "A Down a Right a"; a positional sentence is "a Right a Down a Right a"; the corresponding picture is given by the evaluation of the spatial relations in the positional sentence, from left to right:

a a
a a

Very similarly to Definition 1, the corresponding reverse positional grammar PG′ is:

(1) S := S Up A
(2) S := A
(3) A:= a Left a

The positional sentence becomes now "a Left a Up a Left a" and produces the same picture as abovestarting from the 'a' in the lower right.

Note that to reverse a positional grammar, it is not enough to reverse the right side of the productions. Every spatial relation must also be substituted with a semantically opposite spatial relation. In this example, "Down" becomes "Up" and "Right" becomes "Left".

5.2 Pictorial LR parsing

The generalization of LR parsing to the two-dimensional case has already been treated in (Costagliola *et al.*, 1991, 1992, 1993), where classes of pSLR, pLALR and pictorial generalized LR languages have been characterized.

The parser generation methodology from positional grammars is very similar to the traditional LR technique. The only difference regards the handling of the spatial relations.

As an example, the item "A:= a .Left a" means now that a new 'a' is expected to the left of the just seen 'a'. The containing set-of-items will then be associated to the spatial relation Left.

To handle the positional information, the final LR parsing table for a positional grammar has a new column named "pos", besides the traditional "action" and "goto" parts. The underlying automaton will then have a spatial function associated to each state in order to predict the position of the next symbol to parse.

Example 5.3

The parsing tables for PG and PG′ of Example 5.2 are shown in Figure 5.1 and 5.2, respectively.

state	action		goto		pos
	a	$	A	S	
I_0	s4		2	1	
I_1		acc.			Any
I_2	s4	r2	2	3	Down
I_3		r1			Any
I_4	s5				Right
I_5	r3	r3			Down

Figure 5.1

state	action		goto		pos
	a	$	A	S	
R_0	s4		5	1	
R_1	s4	acc.	2		Up
R_2	r1	r1			Up
R_3	r3	r3			Up
R_4	s3				Left
R_5	r2	r2			Up

Figure 5.2

Every spatial relation name in the column "pos" indicates a spatial function that takes in input a position and returns a terminal, if found, or the end-of-input marker $, otherwise. The only exception is 'Any' that always returns $.

The action "accept" is actually a conditional "accept": if all the symbols of the picture have been processed then accept, otherwise reject. This can be done by marking each visited symbol and looking for unmarked ones.

Looking at Figure 5.1, if state I_5 is reached by shifting a terminal 'a' whose position is (i, j), then the next symbol to process is the terminal Down(i, j) in position (i, j-1). Note that Down is the spatial function associated to I_5. In the following, for sake of simplicity, each Cartesian coordinate (i, j) will be associated with a unique index k.

5.3 The 2-D extension

Definition 2 of "joint graph" applies with no modification to the case of pictorial LR parsing. In particular, for the grammars PG and PG′ of Example 5.2, the Jgraph sets are so defined:

$Jgraph(S) = Jgraph(S_0) \cup Jgraph(S_2) =$
$=\{(I_1, R_1), (I_3, R_1)\}$
$Jgraph(A) = Jgraph(A_1) \cup Jgraph(A_3) =$
$= \{(I_2, R_2), (I_2, R_5)\}$
$Jgraph(a) = Jgraph(a_4) \cup Jgraph(a_5) =$
$= \{(I_4, R_3), (I_5, R_4)\}$

The LR parser with an arbitrary starting point algorithm can also be easily adapted to the case of pictorial languages. The only difference is that the forward and backward parsers are not generalized LR parsers as defined in (Tomita, 1991) but pictorial generalized LR parsers, (Costagliola *et al.*, 1992).

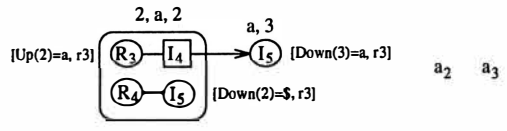
Example 5.4.

Let us consider the grammars PG and PG′ of Example 4.2, the input picture

$a_0\ a_1$
$a_2\ a_3$
$a_4\ a_5$

where each terminal 'a' has been indexed with its position, and i = 2 as the starting position; from the initial stack graph it is possible to reach the following configuration:

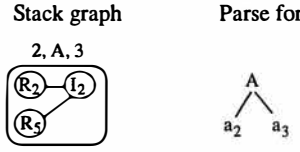
Stack graph Parse forest



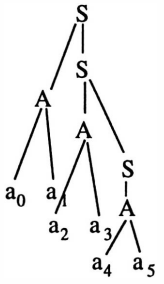
with the joint node ({(I_4, R_3), (I_5, R_4)}, a, 2, 2, a_2_ptr), the simple node (I_5, a, 3, a_3_ptr) and the corresponding [lookahead, action] pairs. The simple stack node with state I_5 has just seen the terminal 'a' in position 3; to take the next action the associated spatial function Down must be applied to the position 3. The returned terminal is 'a' in position 4 and only now the parser can decide a reduction with "(3) A := a Right a". The same explanation can be given for the actions taken by the parsers in the remaining states I_5 and R_3.

The parser with initial state R_4 has no action to take and fails, making the forward parser starting in I_5 fail, too. The only rendezvous operation can then be performed between the parsers starting in R_3 and I_5 on production "A:= a Right a":

Stack graph Parse forest



where the stack graph reduces to the joint node ({(I_2, R_2), (I_2, R_5)}, A, 2, 3, A_ptr). In two other rendezvous operations the picture will be eventually accepted with parse tree:



6 Conclusions

This paper has presented an algorithm to allow LR parsing from an arbitrary starting point of the input. The algorithm is based on Tomita's algorithm and refers to substring parsing as defined in (Rekers — Koorn, 1991). It makes use of two LR parsing tables, one for the original grammar and another for its reverse version.

It can be shown that there is a moderate overhead with respect to the normal parser and that sentences starting with a token that can appear in many different context take more time to parse than sentences starting with a disambiguating token. With respect to substring parsing, no overhead is necessary for the completion of the sentence by any backward or forward parser, as the completion is always determined by the rendezvous operation. Further the Jgraph data structure allows only appropriate reductions cutting on the overhead due to unfeasible reductions.

It is also been showed that, based on the pictorial generalized LR parser in (Costagliola *et al.*,

1992), the extension of the algorithm to the two-dimensional case is immediate.

In this paper, only the simplest form of two-dimensional parsing, the so-called linear pictorial parsing, is referred to. In this type of pictorial parsing, the spatial relations are defined such that the position of the next symbol only depends on the last symbol processed.

More complex forms include the possibility to calculate the next symbol based on the positions of the elements of the last handle or of the whole input so far visited. These forms have been investigated in traditional pictorial generalized parsing and are currently being investigated in the context of LR parsing with an arbitrary starting point.

References

Aho, A.V. — R. Sethi — J.D. Ullman (1985) *Compilers, principles, techniques and tools*. Addison Wesley.

Bossi, A. — N. Cocco — L. Colussi (1983) "A Divide-and-conquer Approach to General Context-free Parsing". in: *Information Processing Letters* 16, 203 – 208.

Costagliola, G. — S.-K. Chang (1991) "Parsing 2D Languages with Positional Grammars". in: *Proceedings of Second Int. Workshop on Parsing Technologies* 235 – 243. Cancun, Mexico, February 13 – 25, 1991.

Costagliola, G. — S.-K. Chang — M. Tomita (1992) "Parsing 2D Languages by a Pictorial GLR parser". in: Catarci, T. & M.F. Costabile & S. Levialdi, (Eds): *Advanced Visual Interfaces*. 319 – 333. Singapore: World Scientific Publishing.

Costagliola, G. — S. Orefice — G. Polese — M. Tucci — G. Tortora (1993) "Automatic Parser Generation for Pictorial Languages". in: *Proceedings of IEEE Symposium on Visual Languages* Bergen, Norway, August 24 – 27, 1993, to be published.

Crimi, C. — A. Guercio — G. Nota — G. Pacini — G. Tortora — M. Tucci (1991) "Relational Grammars and their Application to Multidimensional Languages". in: *Journal of Visual Languages and Computing* 2, 333 – 346. Londra: Academic Press.

Golin, E. J. (1991) "Parsing Visual Languages with Picture Layout Grammars". in: *Journal of Visual Languages and Computing* 2, 371 – 393. Londra: Academic Press.

Helm, R. — K. Marriot — M. Odersky (1991) "Building Visual Language Parsers". in: Robertson, S.P. & G.M. Olson & G.S. Olson, (Eds): *Human Factors in Computing Systems: CHI '91 Conference Proceedings* 105 – 112. Amsterdam: Addison-Wesley.

Rekers, J. — W. Koorn (1991) "Substring Parsing for Arbitrary Context-Free Grammars". in: *Proceedings of Second Int. Workshop on Parsing Technologies* 218 – 224. Cancun, Mexico, February 13 – 15, 1991.

Steel, S. — A. De Roeck (1987) "Bidirectional Chart Parsing". in: *Proceedings of AISB-87* Edinburgh: Scotland.

Stock, O. — R. Falcone — P. Insinnamo (1989) "Bidirectional Charts: a Potential Technique for Parsing Spoken Natural Language" in: *Computer Speech and Language* 3, 1989.

Tomita, M. (1985) *Efficient Parsing for Natural Languages* Boston MA: Kluwer Academic Publishers.

Tomita, M. (1991) *Generalized LR Parsing*. Norwell MA: Kluwer Academic Publishers.

Wittenburg, K. (1992) "Earley-style Parsing for Relational Grammars". in: *Proceedings of IEEE Workshop on Visual Languages* Seattle, USA, September 15 – 18, 1992.

Wittenburg, K. — L. Weitzman — J. Talley (1991) "Unification-based Grammars and Tabular Parsing for Graphical Languages". in: *Journal of Visual Languages and Computing* 2, 347 – 370. Londra: Academic Press.

Woods, W. A. (1982) "Optimal search strategies for speech understanding control". in: *Artificial Intelligence* 18, 295 – 326.

A New Transformation into Deterministically Parsable Form for Natural Language Grammars

Nigel R. Ellis, Roberto Garigliano and Richard G. Morgan

Artificial Intelligence Systems Research Group,
School of Engineering and Computer Science
University of Durham, UK. DH1 3LE
email: {N.R.Ellis|Roberto.Garigliano|R.G.Morgan}@durham.ac.uk

Abstract

Marcus demonstrated that it was possible to construct a deterministic grammar/interpreter for a subset of natural language [Marcus, 1980]. Although his work with PARSIFAL pioneered the field of deterministic natural language parsing, his method has several drawbacks:

- The rules and actions in the grammar / interpreter are so embedded that it is difficult to distinguish between them.
- The grammar / interpreter is very difficult to construct (the small grammar shown in [Marcus, 1980] took about four months to construct).
- The grammar is very difficult to maintain, as a small change may have several side effects.

This paper outlines a set of structure transformations for converting a non-deterministic grammar into deterministic form. The original grammar is written in a context free form; this is then transformed to resolve ambiguities.

1 Introduction

The term *deterministic grammar* is used to refer to a grammar which can be parsed deterministically using a specific parser. The work of [Marcus, 1980] has been extended in the past [Berwick, 1983, Stabler, 1983], but both of these still follow the same method in that the deterministic grammar produced is hand written and therefore difficult to generate, expand and maintain.

Deterministic parsers have three fundamental features. These features appear as constraints in the parsing mechanism and are part of the parsers' structure. The parser has to have a *constrained lookahead* facility. It has to be *data driven* or *bottom-up* to some extent, but also must have the ability to reflect expectation based upon the constructs already formed. All constructs produced from the input to the parser must be part of the output; thus no structure is created and then later destroyed. In a generic non-deterministic parser, when two (or more) grammar rules have identical start symbols, a lookahead must be used by the parser to decide which grammar rule to apply. The use of a lookahead relies upon the following principle:

> "If there the input matched so far forms part of a rule A then some token α will be present in the input. However if the the input forms part of a rule B, then a token β will be present in the input. This process can be extended for similar looking grammar rules."

A parser which uses a lookahead scheme will pause at such objects and then use a lookahead to distinguish between them. To do this, a further stream of symbols is parsed, up to some fixed length (usually denoted by k). Eventually the

parser will reach a point at which it becomes certain of the category of the original symbol. Once this point is reached, the parser will backtrack, allocate this category and continue. This means that the parser will generate the structure for several items more than once, which is an undesirable feature.

2 State of the Art

2.1 Marcus Parsers

2.1.1 PARSIFAL

The major work in the field of deterministic natural language parsing is *PARSIFAL* which is based upon a psychological model of how humans parse language and Marcus' *determinism hypothesis.*[1] *PARSIFAL* has two major data structures — a stack called the *active node stack* and a *lookahead buffer* containing 5 cells (of which only 3 cells can be accessed at any time), which is used to hold grammatical constituents. The *lookahead buffer* processes words in the first input cell based upon the contents of the remaining two cells and therefore can deduce what type of language component it has found. The use of these two data structures ensures that *PARSIFAL* operates in both a top-down and bottom-up fashion. The stack has parents looking for children — a top-down process; the buffer has children looking for parents — a bottom-up process.

PARSIFAL's grammar was designed to capture the generalisations of generative grammar and the structure of constructs which come from Chomsky and Winograd's differing theories of *annotated surface structure.* The grammar consists of pattern/action rules grouped together into units called *packets.* Each packet of rules represents the structure which the parser is attempting to build. Each rule has a numerical priority which is used to decide between rules when more than one pattern matches. Patterns are matched on cells located in the buffer, the *current active node* and the *current cyclic node.* Actions can consist of operations to push or pop nodes from the stack and to activate/deactivate packets of rules. *PARSIFAL* also contains special rules called *attention shift rules* which are used to shift the context of the parser from parsing one constituent to parsing another.

2.1.2 Problems with the Marcus approach

The main problem with Marcus style parsers is that the grammar is encoded in a procedural form which specifies some actions upon a virtual machine. This makes the grammar harder to understand than a grammar written in a declarative manner. Also, because of the procedural form, it is very difficult to expand a grammar, as a change may cause side effects. It is difficult to see the effect of a change in the grammar because the whole of the grammar can have other active packets of rules at the same time as the rule being changed; the recent change may cause some unintential interaction between these rules, rendering a meaning to the grammar which might have been unintentional. Another problem with Marcus style parsers is that they are unable to analyse globally ambiguous grammars in a deterministic manner; when a Marcus parser encounters a fragment of grammar which is globally ambiguous, it marks the built parse tree in a special way. If another interpretation is required for the input sequence, the input has to be completely re-parsed and the initial parse tree is used to guide the parser onto a different interpretation.

2.1.3 Other Marcus Parsers

Several other parsers have been produced as a result of the work of Marcus. However, all of these parsers follow the same basic structure as *PARSIFAL* and therefore share all the drawbacks of the approach. These were: *ROBIE* [Milne, 1986] which looked especially at lexical ambiguity, *L.PARSIFAL* [Berwick, 1983] which was used for grammar acquisition, *YAP* [Church, 1980] which was a modified form of *PARSIFAL* implemented using a finite state machine, *PARAGRAM* [Charniak, 1983] which looked at the parsing of ungrammatical sentences, and *FIDDITCH* [Hindle, 1983] which was used to investigate the sublanguage of military style speech.

[1] Briefly stated, this says : "the syntax of any natural language can be parsed by a mechanism which operates ***strictly deterministically*** in that it does not simulate a non-deterministic machine."

2.2 LR Parsers

2.2.1 Outline

The term LR(k) [Knuth, 1965] is shorthand form for a parser which performs left-to-right parsing building the right-most derivation in reverse (i.e. bottom-up) using at most k terminal symbols as lookahead. LR(k) parsers can only be constructed for unambiguous context-free grammars.

Although natural language grammars are inherently ambiguous, several attempts have been made to apply LR(k) (and the restricted form of LALR) parsing to natural language problems [Shieber, 1983, Pereira, 1985, Briscoe, 1987].

LR parsers are members of the class of shift-reduce parsers [Aho and Johnson, 1974] which are a very general type of bottom-up parser. All shift-reduce parsers incorporate a *stack* for holding constituents as they are built during the parse and have a *shift-reduce table* of *states* and *actions* for guiding the parser. This table contains two types of actions: the *shift* operation, which transfers the next word from the input buffer onto the stack, and the *reduce* operation, which replaces several elements on the top of the stack with a new element.

Shieber and Pereira's work concentrated on using the Unix parser generator, *yacc* [Johnson, 1978], to produce an LALR parser and to use this for parsing natural language. In order to do this, they created several strategies for converting ambiguous context-free grammars into deterministically parsable form. These strategies were based upon semantic rules which exploit basic properties of the English language such as preferences for propositional attachment. Briscoe has also attempted to use LR(k) parsing for natural language. He has concentrated on producing an interactive deterministic parser which corresponds to a specific type of LR(k) parser.

2.2.2 Problems with this approach

The problem with the LR(k) parsing approach is that in order to make a decision, the parser needs to analyse both the left and the right contexts. For some sentences it may be that the size of left and right contexts required to correctly analyse the sentence is as large as the sentence itself. However, LR(k) parsers are restricted to looking at most k symbols ahead. Therefore an LR(k) parser will not be able to analyse a sentence deterministically if the right context required is more than k symbols in length.

Also, since the left context is encoded deterministically into a parse table, any grammar rule which matches the same left hand context and lookahead will cause a *shift-reduce* conflict. This renders pure LR(k) parsing impossible. Shieber and Pereira introduce two rules to solve this problem:

1. Resolve shift-reduce conflicts by shifting.
2. Resolve reduce-reduce conflicts by performing the longer reduction.

Although these rules solve many of the problems, Shieber and Pereira admit that there are several cases in which their parser will not produce an evaluation. For example, the sentence

The horse raced past the barn fell.

causes a reduce-reduce error before the last word.[2] The parser of [Briscoe, 1987] employs a different approach because it can interact with a semantic component which decides which action to perform when facing with a reduce-reduce or shift-reduce conflict. The success of this method relies heavily upon the amount of semantic knowledge recovered from the successfully parsed input.

Although each of these methods partly solves the problem of ambiguity, it should be noted that the action of either parser could at some stage degenerate into an ad-hoc strategy. The parser would then no longer operate in strictly deterministic manner and may have to backtrack.

3 A new approach

The introduction of our transformation algorithm provides the facility for the automatic generation of a deterministic parser from a source grammar given in context free form. A grammar description written in a context free form is far easier to maintain and understand than one written in a procedural format such as Marcus' parser *PARSIFAL* (written in the language *PIDGIN*). The

[2] This is because the finite verb form of 'raced' will be chosen in preference to the participle form.

presence of commands like *create*, *drop*, etc. in a *PIDGIN* grammar make it very difficult to see exactly what language the grammar defines. Moreover, such parsers are difficult to write, maintain and expand, as the effect of making a change in one portion of the grammar may affect another. The results of changes cannot be realized until the parser is thoroughly tested.[3] LR(k) style parsers are also unable to deal with the type of ambiguity present in natural language grammars. Although several extensions to the basic parsing algorithm have been proposed, none of these completely solve the problem.

In the our approach, when some changes are required to the grammar, the original source form is modified and transformed again to produce a new version of the parser. Working in this way ensures that the maintenance and expansion of the grammar does not suffer from the disadvantages of the Marcus system. The transformation system also has the advantage over LR(k) style parsers in that no lookahead is required to parse such grammars; the parser only needs to examine the current input symbol in order for a decision to be made.

4 Notation

In this section, the notation used in the remainder of the paper is introduced.

4.1 Trees

Each tree diagram presented in this paper will consist of a combination of *and* and *or* nodes. *And* nodes will be labelled with a category, *or* nodes (marked with a '+') will remain unlabelled.

Fig. 1: An *and* node.

Fig. 2: An *or* node.

Figure 1 shows a sample *and* node labelled A for the symbols a and b which represents the production rule $A \rightarrow a\ b$, and Figure 2 shows a sample *or* node for the symbol b or the symbol A representing the production rule $O \rightarrow b \mid A$. If any tree diagram has a label of the form $nt = N$, then this represents a node in the tree with name N, which can be referenced by the unique nonterminal name nt. If the name of a node is repeated and appears as a leaf node in a tree, this represents a cycle or repetition of some previously shown item.

4.2 Message passing

If a grammar contains local ambiguity, a parser will normally have to look ahead a number of symbols in the input stream to decide which parsing rule to apply. Rather than using a lookahead, a *dummy* value will be allocated for the name of the parsing rule applied, until more of the input has been parsed and the correct name of the rule determined. Dummy nodes are represented as D in the tree diagrams. Whenever the name of an *and* node has been replaced by a dummy node, the name is moved and attached to the righthand descendant of the node. For example, consider the following grammar G_1:

$S_1 \rightarrow$ A | B
A $\rightarrow$ a b
B $\rightarrow$ a c

This grammar is shown in tree form in Figure 3 and in an equivalent transformed form in Figure 4. Note the messages within the square brackets attached to the nodes b and c.

[3] This task alone may be hard as there is no formalism available for Marcus parsers, so no formal testing methods can be applied.

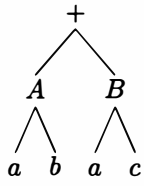


Fig. 3: Tree for grammar G_1.

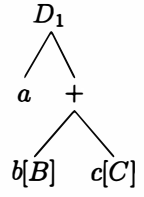


Fig. 4: Transformed version of G_1.

When the parser encounters a dummy node in the grammar, it cannot be sure of the real name of the node, as this node represents some ambiguity which existed in the original grammar. The parser proceeds to match the input symbols against the grammar. When a node is matched which contains messages, the messages are passed back up the built parse tree until a dummy node is found. This dummy node is then replaced by the message at the front of the list of messages found. The search is then continued with the remaining list of messages.

4.3 Gated *or* nodes

When two grammar rules have been unified by a transformation, a single grammar rule is produced which will have a chain of dummy nodes corresponding to the names of the *and* nodes in the two original grammar rules.

This new grammar rule preserves the structure of the original two grammar rules by using a special type of *or* node called a *gated or* node. These nodes prevent the parser from following a path in the new rule which is a mixture of the original two grammar rules.

Consider the right child of D_1 shown in Figure 6. This type of *or* node will be referred to as a *gated or* node. The values in the braces are tests on the name of the left child of D_1. For example, if the parser had matched the input sequence *ab*, then the dummy node D_2 would have been replaced by the message A and the parser would choose the path below the gated node $\{A\}$. Likewise, if the parser had matched the input sequence *ac*, the dummy node D_2 would have been replaced by the message B and the parser would choose the path below the gated node $\{B\}$.

$S_2 \rightarrow$ E | F
A $\rightarrow$ a b
B $\rightarrow$ a c
E $\rightarrow$ A d
F $\rightarrow$ B e

Fig. 5: An example grammar G_2.

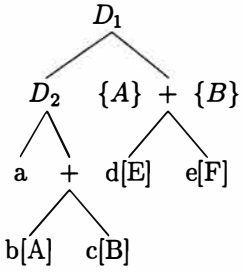


Fig. 6: Transformed version of grammar G_2

4.4 Special gated *or* nodes

In order that the structure of the grammar is preserved when the parser is following a cycle in the transformed grammar, it must have some method of recording the name of the cycle which it has followed previously. This mechanism is implemented by the use of *cycle markers* and *special gated or nodes.*

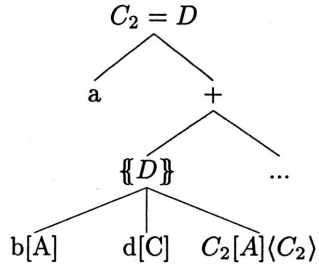


Fig. 7: Grammar G_3

Cycle markers are represented by angled brackets e.g. $\langle C_1 \rangle$ and special gated or nodes represented by double braces e.g. $\{\{C_1\}\}$. When the parser encounters a special gated or node, the

name of the current cycle being followed is compared with the values contained in the node. If a match is found, then the parser continues by following the descendants of the gated node. An example of a gated or node and a cycle marker may be found in Figure 7 (taken from Figure 12).

There will always be one value D in a collection of special gated or nodes which represents the path the parser should follow if it has not already followed a cycle.

5 Transformation Method

The transformation into deterministic form is performed by the algorithm given in this section. The transformation is divided into three stages: pre-transformation, main transformation and post-transformation. The pre-transformation stage prepares the grammar for processing by the main transformation by removing any previous unification from the source grammar. The main transformation unifies any ambiguity which may exist in the source grammar and the post-transformation tidies the transformed grammar making it suitable for input into the parser. A detailed example of the transformation is also given.

5.1 Pre-Transformations

The pre-transformation unpacks any unification which may exist in the source grammar.

1. Lift the left-most *or* nodes above the *and* nodes, removing any empty nodes which may be present. This step continues the unpacking of any previous unification.
2. Flatten any chains of *or* nodes into one *or* node.
3. Repeat steps 1 to 2 on whole graph until the transformations can not be applied.

5.2 Main Transformation

For each *or* node O in the grammar with descendants $t_1 \ldots t_n$, do the following:
Take the first descendant t_1 and compare it with all the others. If a matching descendant t_i $(1 < i \leq n)$ is found (following the method below), unify the two and start again comparing the resulting tree to the rest. If no match is found, leave t_1 in the *or* node and repeat the procedure for the rest of the descendants. The comparison process is as follows:

5.2.1 Comparison between two graph segments:

1. For each tree t_1 and t_i, list the sequences L_1 and L_i of leftmost nodes from the root node to the leftmost terminal node;
2. If no common nodes are found in the lists, the two trees cannot be unified by this algorithm;
3. If a common node c is found, mark it;
4. Count the number of nodes n_1 and n_i from the first node to the common node c in L_1 and L_i;
5. If $n_1 \neq n_i$, add dummy *and* nodes to the top of the shortest tree (t_1 or t_i). The existing tree is the left child, and an empty node is the new right one. A dummy message is then added to the empty node. Any messages carried by the previous node are passed to the new one. When two trees are unified, the resulting tree must be complex enough to accommodate the more complex of the two, so the shorter tree must be balanced to match the larger one.

5.2.2 Make the unified tree:

Given two balanced trees t_1 and t_i with a common node c in the list of nodes from the root to the leftmost node, do the following:

1. Take the number of nodes above c and create that number of dummies. Each of these dummies will have an *or* branch gate as the right child.
2. Put c as the leftmost of the chain. This is done because the node c represents the common elements of both trees.
3. Put a special gated *or* node, S, as the sibling of c. This node represents the first node of each of the trees being unified, and is an *or* node because all structures above this node are different.

4. Attach to S a gated node with the name D. Add to it an *or* node containing the siblings of c in each of the two trees t_1 and t_i. These represent the possibilities which can follow from c in the two trees being unified before the name of any cycle has been resolved.

5. For each cycle C in t_1 or t_i attach to S a special gated node with name C. Add to this node, the siblings of c from the tree the cycle appears in. These nodes represent the choices available in the grammar if the parser is following a specific cycle name.

6. If any gated nodes are repeated in S, remove the duplicates, adding the possibilities below each duplicate to the remaining node in S.

7. Add the name of the parent of the siblings in each of the original trees to the message list of each sibling. This allocates the messages which will be passed to the dummy nodes.

8. Add to each branch gate (sibling node of the dummy chain) the name of the original left sibling (if more than one) and the possible choices which follow from it. This ensures that the messages passed up to the dummy nodes are used to lead the parser to the correct possibility at an *or* node.

9. Repeat this transformation on the new *or* node, after having flattened it.

5.3 Post-transformations

The post-transformation tidies the transformed grammar to make it suitable for input to the parser.

1. Flatten the chain of *or*s which do not carry messages;

2. Unify the same gates under an *or*, thus making the gates disjoint; If there is a gated *or* node which has two (or more) gates which contain the same message, the common message is removed from each gate and a new gate with this message is formed which has an *or* node as its child. This *or* node has each of the common possibilities as children.

3. If an *or* node contains an empty node as a descendent, then place this node at the end of the list of descendents. This ensures that there is no backtracking.

5.4 Example

An example application of the transformation algorithm to the grammar G_3 is shown in Figures 9 – 12 on the next page

$S_3 \rightarrow$ B | E
A $\rightarrow$ a O_1
B $\rightarrow$ A c
C $\rightarrow$ a d
E $\rightarrow$ C O_2
$O_1 \rightarrow$ b | A
$O_2 \rightarrow$ e | E

Fig.8: Grammar G_3

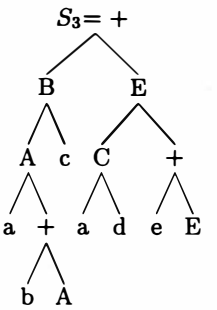


Fig.9: Start grammar G_3.

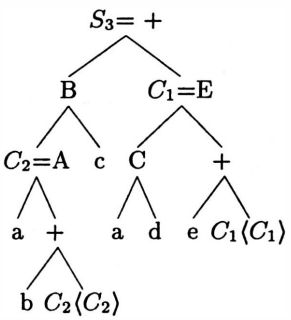


Fig.10: Mark cycles.

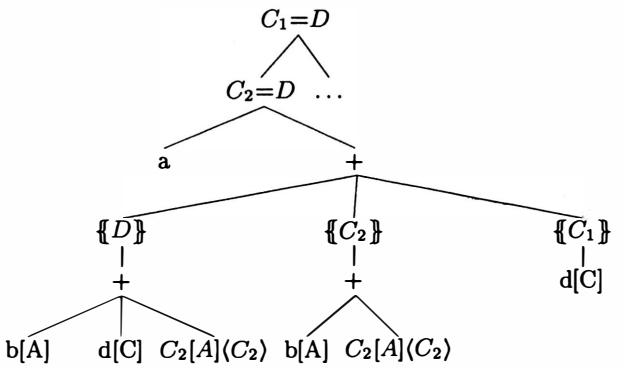


Fig.11 : Unify: build dummy chain and special gated nodes.

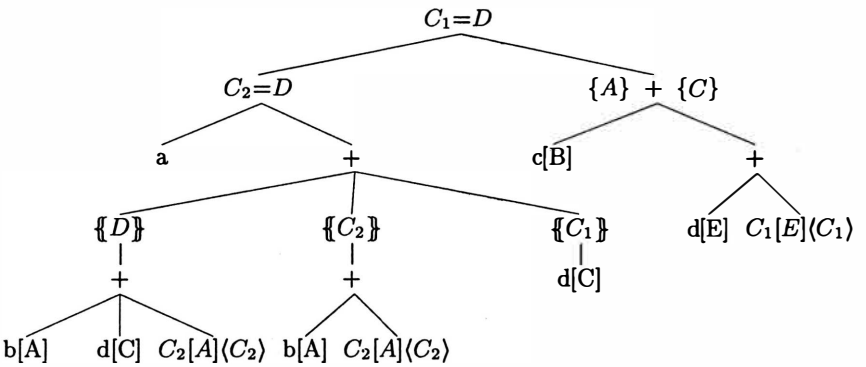


Fig.12: Unify: build gated *or* nodes.

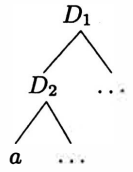
5.5 Parsing method

Initially the parser operates in a top-down fashion, matching the input from left to right. The parser checks the first input symbol against all of the possibilities below the top *or* node in the transformed grammar. If a match is found, then the parser proceeds to match the input seqence by following that possibility. The parser continues in this way for each *or* node encountered in the grammar until one of the following possibilities occurs:

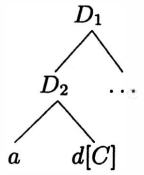
- The parser matches a node which contains messages. These messages are then passed back up the parse tree to replace the dummy nodes encountered whilst building the tree.
- The parser encounters a dummy node which has an empty node as its right child. The parser then replaces the dummy node and right child with the left child. This situation occurs when a tree has been balanced for unification.
- The parser reaches a gated *or* node. The parser then follows the possibilities below the gated node which contains the name of the message which replaced the dummy sibling node of the gated *or* node (gated *or* nodes always appear as a sibling node to a dummy node in the transformed grammar).
- The parser encounters a special gated *or* node. If the parser has not yet resolved the name of any cycles, the path below the dummy gate is followed. If the name of the cycle has been resolved, the path below the gate containing the cycle name is followed. Cycle names are resolved by nodes with angle brackets. For example, the node $C_1[m_1 \ldots m_n]\langle R\rangle$, represents a cycle named C_1 with messages $m_1 \ldots m_n$ whose real cycle name has been resolved to the R.

Below is an example parse of the transformed grammar G_3 shown earlier in Figure 12. The grammar G_3 presented earlier can match the input sequences $a^n bc$ and $(ad)^n e$ where $n > 0$. If the input given to the parser is *adade* (for $n = 2$), then the following actions will be performed (the symbol | represents how far the input has been parsed).

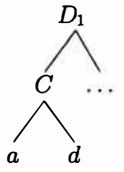
Input: $|adade$
Action: Match symbol a, build chain of dummy nodes.



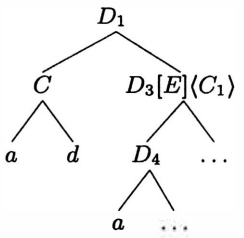
Input: $a|dade$
Action: Cycle name is unresolved, so follow the path below the dummy gate $\{\!\{D\}\!\}$ (sibling of a). Match the symbol d.



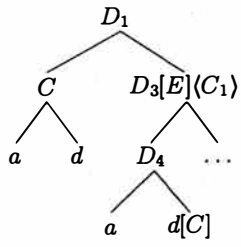
Input: $ad|ade$
Action: Pass message C back up to replace the dummy node D_2.



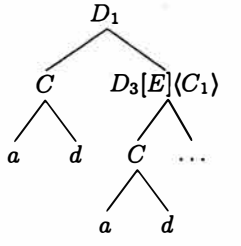
Input: $ad|ade$
Action: Match the gated node $\{C\}$ as its sibling node is now C. No symbol e, so follow the cycle C_1 with message E and cycle name resolved to C_1. Now match the symbol a.



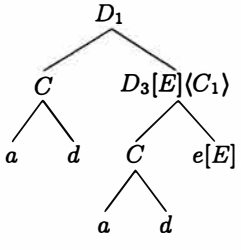
Input: $ada|de$
Action: The cycle name has been resolved to C_1, so follow the path below the special gated node $\{\!\{C_1\}\!\}$. Match the only possibility of d with message C.



Input: *adad|e*
Action: Pass message C back up to replace the dummy node D_4.

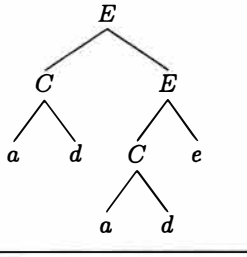


Input: *adad|e*
Action: follow the path below the gated node $\{C\}$ as its sibling node is C. Match the input symbol e with message E.



Input: *adade|*
Action: Pass message E back to replace dummy node D_3 and the second message E to replace the dummy node D_1.

Parse is now finished.



6 Improvements

Several improvements are planned to the transformation algorithm. These include adding a facility to deal with homonymy.[4] In addition lookahead gates can be added to *and* nodes under an *or* node to prevent the parser needlessly descending a chain of nodes to match the left-most symbol. Other improvements which could be made involve expanding the algorithm to deal with features.

7 Conclusion

In this paper, we have outlined a transformation for converting a non-deterministic context free grammar into deterministic form. A complete formalism of the transformation algorithm has been produced. This is discussed in [Ellis *et al.*, 1993]. Work is also in progress to produce a method for transforming globally ambiguous grammars.

The transformation outlined has been implemented in the lazy functional language Miranda[5] and has been applied to the large natural language processing system LOLITA [Garigliano *et al.*, 1993]. LOLITA is a general natural language (English) tool which has been under development at the University of Durham for the last four years. The LOLITA system is built around a large semantic network which holds knowledge that can be accessed, modified or expanded using natural language input and has a grammar of some 1600 rules. The system can parse complex text (such as newspaper articles), semantically and pragmatically analyse its meaning and add relevant information to the network. The system can also answer natural language interrogations about the knowledge held in the network by generating natural language from the network representation.

Acknowledgements

The authors would like to thank Greg Lee of the University of Hawaii for the production of the tree drawing package used in the production of this report. Nigel R. Ellis is supported by a grant supplied by the Science and Engineering Research Council of Great Britain.

[4] For example, the word 'bank' is homonymous as it can represent either a noun or a verb. If the transformation can be extended to deal with homonymous words such as this then the parsing of transformed grammars can be made more efficient.

[5] Miranda is a trademark of Research Software Ltd.

References

A. Aho and S. Johnson, "Programming Utilities and Libraries; LR Parsing", *Computing Surveys*, 4(6):99 – 124, June 1974.

R. Berwick, "A deterministic parser with broader coverage", in *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 710–712, 1983.

E. Briscoe, *Modelling Human Speech Comprehension*, Series in Computer Science, Ellis Horwood, 1987.

E. Charniak, "A Parser with Something for Everyone", in M. King, editor, *Parsing Natural Language*, chapter 7, pages 117–149, Academic Press, London, 1983.

K. Church, "On memory limitations in natural language", Unpublished Masters thesis, Laboratory for Computer Science, MIT, 1980.

N. Ellis, R. Garigliano, and R. Morgan, "A Transformation Algorithm for Converting Non-Deterministic Grammar into Deterministic Form", Technical Report 4/92, Artificial Intelligence Systems Research Group, School of Engineering and Computer Science, University of Durham, UK, 1992.

N. Ellis, R. Garigliano, and R. Morgan, "A Language for defining transformations on graph grammars: definition and use.", Technical Report ?/93, Artificial Intelligence Systems Research Group, School of Engineering and Computer Science, University of Durham, UK, 1993.

R. Garigliano, R. Morgan, and M. Smith, "The LOLITA System as a Contents Scanning Tool", in *Proceedings of the 13th International Conference on Natural Language Processing, Avignon, France*, May 1993.

D. Hindle, "Deterministic parsing of syntactic non-fluencies.", in *Association for Computer Linguistics*, pages 123 – 128, June 1983.

S. Johnson, "*yacc*: Yet another compiler-compiler", Technical report, Bell Laboratories, Murray Hill, New Jersey, USA, July 1978.

D. Knuth, "On the translation of language from left to right", *Information and Control*, 8(1):607–639, 1965.

M. Marcus, *A Theory of Syntactic Recognition for Natural Language*, MIT Press, 1980.

R. Milne, "Resolving lexical ambiguity in a deterministic parser", *Computational Linguistics*, 12(1):1–12, 1986.

F. Pereira, "A new characterization of attachment preferences", in D. Dowty, L. Kartunnen, and A. Zwicky, editors, *Natural language parsing: psychological, computational and theoretical perspectives*, pages 307–319, Cambridge University Press, 1985.

S. Shieber, "Sentence disambiguation by a shift-reduce parsing technique", in *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 113–118, Cambridge, Mass., June 1983.

E. Stabler, "Deterministic and Bottom-up Parsing in PROLOG", *American Association for Artificial Intelligence*, 1983.

A Principle-based Parser for Foreign Language Training in German and Arabic*

Joe Garman* Jeffery Martin* Paola Merlo† Amy Weinberg*

* Dept. of Linguistics and Institute for Advanced Computer Studies
University of Maryland at College Park, U.S.A.
email: {garman|jeffmar|weinberg}@umiacs.umd.edu

† Dept. of Linguistics and Dept. of Psychology
University of Geneva, Switzerland
email: merlo@divsun.unige.ch

Abstract

In this paper we discuss the design and implementation of a parser for German and Arabic, which is currently being used in a tutoring system for foreign language training. Computer-aided language tutoring is a good application for testing the robustness and flexibility of a parsing system, since the input is usually ungrammatical in some way. Efficiency is also a concern, as tutoring applications typically run on personal computers, with the parser sharing memory with other components of the system. Our system is principle-based, which ensures a compact representation, and improves portability, needed in order to extend the initial design from German to Arabic and (eventually) Spanish. Currently, the parser diagnoses agreement errors, case errors, selection errors, and some word order errors. The parser can handle simple and complex declaratives and questions, topicalisations, verb movement, relative clauses — broad enough coverage to be useful in the design of real exercises and dialogues.

1 Introduction

This paper describes the design and implementation of a parser for German and Arabic, which is currently being used in a tutoring system for foreign language training. Computer-aided language tutoring is a good application for testing the robustness and flexibility of a parsing system, since the input is usually ungrammatical in some way. The system is portable in that we extend the initial design for German to handle Arabic and (eventually) Spanish. Efficiency is also a concern, since tutoring applications typically run on personal computers, with the parser sharing memory with other components of the system. Our system is principle-based, which ensures a compact representation.

Robustness is required because the system must be able to parse incorrect input. Since the user is not a native speaker of the language, the system must handle spelling errors such as (1a) as well as grammar errors, as in (1b).

(1) (a) *Der Fluß flest auf Westen*
The river flows to west

(b) *Der Fluß fließen auf des*
The river flow.pl to the
Westen
west.gen
'The river flows to the west'

*The names of the authors are in alphabetical order. This paper is the result of cooperative work in all its stages. However, as far as legal requirements are concerned, Paola Merlo takes responsibility for sections 4 and 5.

(2) (a) *at-tariiqu* *taqa9u*
road.sg.def locate.3.f.sg

(b) *aT-Tariiqu* *taqa9aayna*
road.sg.def locate.3.f.pl
'The road is there.'

The system must be flexible because it must be able to handle different levels of ill-formedness. For example, it must be able to detect the difference between (1a) and (1b). In the former the verb *fließt* is incorrectly typed, while in the second sentence the verb is incorrectly inflected and the wrong case is selected for the prepositional phrase, which should be *auf Westen*. The sentences in (2) illustrate a similar pair from Arabic. In (2a) the word for *road* should have an emphatic consonant (signified by a capital letter). In (2b), there is an agreement error (the subject is third feminine singular, while the verb is third feminine plural). The former type of mistake must be corrected for the parser to succeed. This correction is done by an interaction with the user. The latter type of error is bypassed by a special mechanism of defaults that enable the parser to analyse some kinds of incorrect input.

Portability in a language processor means that the design can be reused for different languages without fundamental changes. The dictionaries and some of the grammatical information will be different, but it is desirable to design the Natural Language Processing (NLP) component in the most general way.[1] Arabic and German are good bounding cases to test the portability of the design. The languages are similar in having rich inflectional morphology and in having various forms of grammatical agreement. They are different enough to constitute a fair test of the portability of the system. For instance, they have different word orders: German is basically verb-final, with movement of the verb in second position in main clauses, while Arabic has SVO word order, with VSO occurring frequently in surface order. Grammatical subjects are usually dropped in Arabic and expressed by agreement with the verb, while in German null-subject sentences are not grammatical. Small clauses are extremely common in Arabic, and usually copular verbs are not expressed, while German sentences present a clear distinction between predicates (verbs) and arguments (nouns). Thus, for a parser to be applicable to both languages, it must make use of very abstract properties of the grammar.

Finally, the system must be efficient. Ideally, it must be able to parse a sentence, detect the errors and produce an output in the same amount of time it would take a human tutor to perform the same task, otherwise the time lag would decrease the student's attention and motivation. Computational efficiency must not be bought at the expense of space compactness though, because the system must fit on a PC with many other programs running at the same time.

The principle-based approach to parsing allowed us to meet all of these requirements. Previous formalisms, for example, the EST version of generative grammar (Chomsky 1965, 1973, 1977 among others) assumed that every construction of a language was syntactically represented by a grammatical rule. Thus big, monolithic grammars needed to be stored and consulted for any parser with reasonably wide linguistic coverage. Recent developments in grammatical theories, in many different frameworks, have succeeded in identifying the unifying principles of many, apparently unrelated, linguistic phenomena.

The Government-Binding (GB) framework (Chomsky 1981) provides construction-independent principles that are grouped into interacting modules. The modules are parameterised, so that by modifying them to a small degree, one can generate the patterns associated with a variety of languages. The modularity of the grammar makes it easy to relax certain constraints and thereby obtain a parse for various types of ungrammatical input. Because language-dependent information is separate from language-independent information, the same parsing design is valid across languages. Finally, the factorization into modules makes the grammar compact, and consequently storage needs are minimised.

Our implementation provides experimental answers to the issues of parsing ill-formed input, generalisation across languages, and compactness of representation. It addresses the problems of modularisation, how much compilation across principles is necessary, and how various principles can be efficiently interleaved. The following sections discuss the actual implementation, illustrate the design criteria, and finally

[1] In the best case the design is explicitly parameterised so that it specifies a family of parsers.

evaluate the performance of the parser. The part of this research which addresses issues related to foreign language training and tutoring systems is mentioned only in discussing the constraints on parsing design imposed by the application. Section 2 provides an overview of the NLP system, while sections 3, 4, and 5 illustrate the routines for the recovery of phrase structure, feature annotation, and error detection, respectively. Evaluation and illustration of coverage are presented in section 6, followed by concluding remarks in section 7.

2 Overview

The Input/Output The input to the parser is an unannotated German or Arabic sentence. Currently the Arabic is in phonetic transcription, but an interface allowing the use of Arabic script is being developed. The output consists of a parse tree which encodes the hierarchical and linear relations between the elements of the sentence (which is passed to a semantic analyser), and a (possibly empty) list of errors in the sentence, which is passed to the tutor. The following examples show sentences with various errors which illustrate the input/output of the parser (omitting the parse trees).

(3)

```
==> Das frauen gestern hat in des berges
geblieben.
Errors: (1) The article "das" does not
agree with "frauen". (2) Word Order: The
verb "hat" should be in second position.
(3) There is a case selection error between
"geblieben" and "in des berges". The verb
"geblieben" is a state verb and takes dative
case. (4) Wrong auxiliary: "hat". The verb
"geblieben" takes sein. (5) The subject "das
frauen" does not agree with the verb "hat" in
person or number.
```

(4)

```
   ==> ayna aqa9u mintaqati d-dibdibbatu ?
Errors: (1) Spelling error: emphatic/non-em-
phatic substitution in "mintaqati" (2) Case
error between "minTaqati" and "d-dibdibbatu"
(3) Verb/Subject agreement error between
"aqa9u" and "minTaqati d-dibdibbatu"
```

These examples show a simple Prolog interface; the tutor interface on the PC is written in an authoring language which is not discussed here. The parse trees are not displayed to the student.

The Components The main components of the parser are a morphological analyser, a syntactic parser, and an error handling facility. The general overview of the system is shown in Figure 1. The parser receives the input sentence and it sends it to an interactive preprocessor which expands possible contractions, such as *zum* into *zu dem*, and checks if the input is correctly typed.[2]

Parsing and morphological analysis perform a single pass on an input sentence. The parser calls the morphological analyser to place words on the parser's buffer stack. Each call to the morphological analyser returns a set containing all the analyses of the next input word, as it may not be possible to determine which among them is correct at the point where the morphological analyser is called. If only one analysis is selected, it may cause the parser to fail in a later state, and require backtracking. The morphological analysers for German and Arabic are somewhat different owing to the differing morphological features of the two languages.

When the morphemes are correctly identified, the word is passed to the syntactic processor. Before entering this stage, each token is projected to its categorial node; for instance, the verb form *fließt* is projected to V. The shift-reduce parsing algorithm operates on a small set of context-free rules, which store only a very limited amount of hierarchical information. Other kinds of information to build the right parse tree and annotate it correctly are stored in a set of constraints that must be satisfied before a rule can apply. We discuss the rules and constraints in the next two sections.

[2]This is the level at which misspellings are corrected. For more details on the preprocessor, routines on lexical search and morphological analyser, see Azadegan *et al.* (forthcoming).

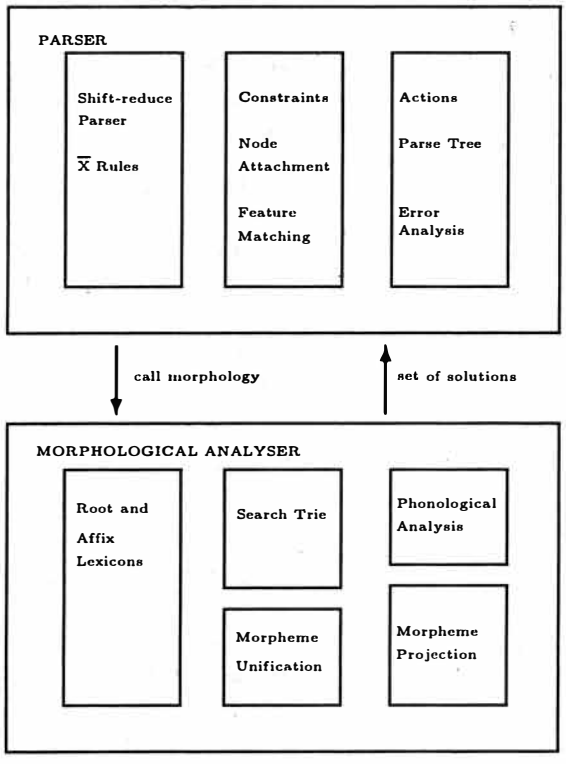


Figure 1: Main Blocks in the Natural Language System

3 The Recovery of Phrase Structure

The parser builds structure by using an augmented shift-reduce method (Aho and Ullman 1972) modified to be more suitable for natural language parsing. Two main modifications are used.

First, the $\overline{X}$ system of our grammar framework is minimal. Most of the work in parsing consists in constraint checking rather than manipulating $\overline{X}$ rules. Thus, these rules constitute a context-free backbone anchoring a set of grammar constraints, similar to implementations based on unification grammars, such as PATR-II (Shieber 1986). To separate constraints from phrase structure, we use rules of the following form:

(5) $X \rightarrow Y\ Z \Leftrightarrow f(X,Y,Z)$

The lefthand side is an $\overline{X}$ production, while the predicate $f(X,Y,Z)$ on the righthand side represents a conjunction of grammar constraints on the features of the nonterminals which must be satisfied to license the production. The form of these constraints and the way they are applied is discussed in the section on constraints.

Second, it is necessary to express movement rules, which relate overt categories to empty categories (also called gaps) in a different position. Our design separates movement rules from $\overline{X}$ rules. For this purpose a *move-x* operation is introduced in the parse cycle. The use of gaps achieves a greater degree of generalisations in the treatment of superficially different constructions, for instance relative clauses and questions. In order to maintain a limited number of $\overline{X}$ rules, however, the computation of gaps is performed

on-line, by specific routines, rather than precompiled into the rules. These two features of the parser are presented in more detail below.

3.1 The $\overline{X}$ Rules

$\overline{X}$ theory is intended to capture universal properties of phrase structure by means of a very small set of rules, which define the phrasal projections and their parts, distinguishing heads from non heads, the latter including complements, specifiers, modifiers, and adjuncts. The $\overline{X}$ rules refer only to the bar level of the phrase; they do not refer to other grammatical features (Chomsky 1970, Jackendoff 1977, among others). $\overline{X}$ rules are maximally binary in our system (Kayne 1984). Although they refer to phrases of a specific level, they do not refer to the specific category or features of the phrases. The complete $\overline{X}$ system employed by the German parser is given by the set of Prolog clauses in (6). The rules for Arabic are given in (7). It is evident from (6) and (7) that very similar sets of rules can handle both languages.

(6) GERMAN:

```
x(2,C)==>[x(2,C1),x(1,C),prespecifier].
x(1,C)==>[x(0,C),x(2,C1),postcomplement].
x(1,C)==>[x(2,C1),x(0,C),precomplement].
x(1,C)==>[x(2,C1),x(1,C),premodifier].
x(1,C)==>[x(1,C),x(2,C1),postmodifier].
x(2,C)==>[x(2,C1),x(2,C),preadjunct].
x(2,C)==>[x(2,C),x(2,C1),postadjunct].
x(1,C)==>[x(0,C),unary1].
x(2,C)==>[x(1,C),unary2].
```

(7) ARABIC:

```
x(2,C)==>[x(2,C1),x(1,C),specifier].
x(1,C)==>[x(0,C),x(2,C1),complement].
x(1,C)==>[x(1,C1),x(1,C),premodifier].
x(1,C)==>[x,(1,C),x(1,C1),postmodifier].
x(2,C)==>[x(2,C1),x(2,C),preadjunct].
x(2,C)==>[x(2,C),x(2,C1),postadjunct].
x(1,C)==>{x(0,C),unary1].
x(2,C)==>[x(1,C),unary2].
```

A term `x(Level,Category)` represents a phrasal projection, where `Level` is the $\overline{X}$ level taking the values {0,1,2}, and `Category` takes as its value an atomic category symbol from the set {N,V,A,P,Adv,Det,Infl,Comp,Conj}. For example, `x(N,2)` is an NP, and `x(V,1)` is the first projection of V. In $\overline{X}$ theory, the nonterminal domain is restricted to projections of the atomic category symbols. This restriction holds in (6) and in (7) since the category of a projection is not instantiated in any of the clauses; all categorial information in the system comes from lexical entries themselves. Each righthand side contains a head, which shares the category variable with the lefthand side; a label, which is used to index constraints and tree-building predicates; and possibly a satellite, which is a specifier, complement, adjunct, or modifier.

This design allows no compilation of bar level and category type, at this stage. Berwick (1991) suggests that this organisation slows down the parser, as found experimentally by Dorr (1987). However, we differ from Dorr in two main respects: the rule set we use is much smaller, and the flow of control is serial. Assuming that the average number of conflicts in a compiled grammar is a good indicator of the amount of nondeterminism that a shift-reduce parser has to face, Merlo (1992) reports comparisons of LR tables derived from grammars which differ only in the instantiation of the $\overline{X}$ rules. Rules like those in (6) and (7) are compared to rules of identical format, where the head of the rule is instantiated, for instance, in our notation, `x(2,N) ==> [x(2,C1),x(1,N)]`. Merlo finds that instantiation of heads expands the grammar, of course, but does not reduce the number of conflicts.[3] This leads us to think that non-determinism is not affected by adding category information to the rules, unless filtering constraints, such as co-coccurrence restrictions, are also added. Co-occurrence restrictions, however, and their linear order, are language-specific. Therefore, we have chosen to check category information on-line, at the same time as categorial co-occurrence.

From the point of view of grammar engineering, this choice has the advantage of keeping the basic types of information distinct: topological (configurations), lexical, and long distance relations, thus reflecting the information structure of the theory more directly.

[3] In fact the number of conflicts per state increases from an average of 4.2 for $\overline{X}$ grammars, to 10.3 for instantiated grammars.

3.2 The Shift-Reduce Algorithm

The parser uses three primary data structures: a main stack, a buffer stack, and a hold store which contains copies of moved phrases. In addition, it takes a token stream (the input) and outputs an annotated tree. The recursive parse procedure thus has five arguments:

(8) parse(Stack, Buffer, Hold, Tree, Stream)

A parse configuration consists of a 4-tuple $(Stack, Buffer, Hold, Stream)$. Since the tree is used only for output, it is not included in the configuration. Various operations including reductions, movement rules, shift, and accept, are used to go from one configuration to the next. It is common in shift-reduce parsing to enter a configuration from which several operations are possible. Our algorithm chooses one operation among those available by giving priority to some operations over others. To choose which operation to apply in a given configuration, the algorithm uses Prolog's control strategy (depth-first search) to select the first operation that matches the configuration. Each operation is implemented by a Prolog clause, and the clauses are ordered as in (9).

(9)
1. morphological analysis
2. accept
3. attention shift
4. move-x
5. reduce X = Y Z (binary reductions)
6. reduce X = Y (unary reductions)
7. pop
8. shift
9. fail

Nondeterminism caused by lexical and structural ambiguity is handled in the current system by backtracking. Despite the worst case time complexity, we have found that this approach gives good performance in practice. We are currently experimenting with the use of a graph-structured stack (Tomita 1985), which allows all parser operations to be computed in parallel.

The parser accepts sentence fragments (NP's, VP's, PP's, and so forth) as well as full sentences.[4] The movement clause determines whether a movement rule should apply in a configuration. This operation is discussed in more detail in the next subsection. Binary reductions are performed by the clause in (10). In this clause, the parser attempts to match the top two elements on the stack with the righthand side of a binary rule of the form:

(10) `x(L,C)==>[x(L1,C1),x(L2,C2),Rule]`

The name of a rule, here indicated by the parameter `Rule`, is used to index the constraints which must apply to the feature sets of the elements on the stack if the rule is to succeed. If a rule matches but the constraints for that rule fail, then an attempt is made to match other rules. If the constraint succeeds, tree-building actions are performed and the parse continues with the reduced phrase on top of the stack. An unconditional shift simply moves the top element from the buffer onto the main stack.

3.3 The Move-x Component

Two basic types of movement rules are implemented. The first involves movement of a maximal projection. The most common instance of this type is movement of a question phrase to the sentence-initial position, as in the German example (11a), and in the Arabic example (11b), where the question word (*ayna*) is also moved to the front of the sentence. This type of movement can also generate topicalisations, clefts, and some relative clauses in both Arabic and German.

(11)

(a) *In welcher Richtung fließt die*
In what direction flows the
Donau?
Danube

'In which direction does the Danube flow?'

(b) *Ayna taqa9u maHaTTatu sh-shurTati*
where located station police
nisbatan li T-Tariqi
relation to road.def

'Where is the police station in relation to the road?'

[4] This is needed for exercises where the appropriate response is a sentence fragment.

The second type of movement involves movement of a level 0 phrase (a head), which occurs in the yes/no question in (12), where the verb *darf* (*must/may*) has moved.

(12) *Darf ich hier bleiben?*
may I here stay
'May I stay here?'

Following Thiersch (1978), the underlying position of the verb in German is the sentence-final position. This is the position occupied by the verb in a subordinate clause, as in (13).

(13) *Ich weiss nicht ob ich hier bleiben darf.*
I know not if I here stay might
'I don't know if I may stay here.'

To form a yes-no question, the verb is first moved to second position, and from there to the clause-initial position. This type of movement is also used frequently in Arabic, where along with the standard Subject Verb Object word order, the Verb Subject Object order is obtained by moving the verb to the front of the sentence. An example is given in (14).

(14) *Taqa9u maHaTTaatu shurTati d-dibdibbati 9alaa Tariiqin*
located station police def.Dibdibba on road
'The Al-Dibdibba police station is located on a road.'

The parser uses the two steps in (15) to relate the surface position of a phrase to its underlying position.

(15)
- Stack-to-Hold Rules: put a trace of the antecedent onto the hold store
- Hold-to-Stack Rules: move a trace from the hold store onto the stack.

Each type of movement rule uses a data structure called the Hold Store (Wanner and Maratsos 1978), which is used for temporary storage of the moved element. The Hold Store contains two cells, one for level 2 phrases (X2HOLD) and one for level 0 phrases (X0HOLD). The use of a Hold store has been criticised from a psycholinguistic and linguistic point of view (Berwick and Weinberg 1984, among others). However, recent work on the connectivity of natural languages (Stabler 1993) suggests that allocating a (small) finite number of *memory registers* to each type of linguistic entity that undergoes long-distance relations captures a wide-spread generalisation, both in syntax (questions, causatives) and morphology (applicatives, datives).

For the Stack-to-Hold operation, we must identify when a phrase is not in its underlying position; *i.e.* we must locate the antecedent. Since the type of positions to which a phrase may move is limited and predictable, the parser consults a table of move-x configurations to determine that a phrase is not in its underlying position.

For instance, the rule applies in a configuration where the stack contains the single *wh*-phrase *in welcher Richtung* ('in what direction') and where the buffer contains the verb *fließt* ('flows'). In such a configuration, a trace of *in welcher Richtung* is placed on the X2 hold store. The trace is identical to the antecedent in all features except the spelling, since the trace does not appear in the surface form.

For the Hold-to-Stack operation, we must find the underlying position of the trace on the hold stack. Again the parser consults a table of configurations which determines whether the trace should be moved from the buffer to the stack. For instance, this rule would apply in a configuration where the verb *fließt* is on top of the stack, and the trace of the phrase *in welcher Richtung* is in the X2 hold store. There is a condition on the rule which requires that the moved phrase be a possible complement of the verb. In this instance, the condition on the rule is satisfied, since *in welcher Richtung* is a possible complement of the verb *fließen*. In this configuration, the trace is moved from the hold store to the second position on the stack. It is moved to second position because complements in German occur on the left of the verb.

We should underline once again that we have adopted a direct implementation of the principles of the grammar. Other approaches (Dorr 1993, Fong 1991) precompile the possible positions where an empty category can occur. Using slash rules (Gazdar *et al.* 1985), in the $\overline{X}$ schema would amount to the same compilation. Clearly, our approach can lead to frequent postulation of

empty categories in many positions that are not licensed by the grammar.[5] Thus, this method of recovering phrase structure information must be coupled with filtering constraints that weed out incorrect derivations as soon as possible. We illustrate the solution adopted for this problem of interleaving the principles in the next section.

4 Constraints

As a result of using an $\overline{X}$ backbone to parse, most of the feature percolation and feature annotation that could be encoded in the nonterminals is performed in this system by constraints on attribute annotation associated with each $\overline{X}$ rule. The set of constraints is partitioned into (non disjoint) subsets that are indexed to each rule, and must be satisfied for the rule to apply. For instance, the following rule is indexed into the set of constraints by the index `postcomplement`.

(16) `x(1,C)==>[x(0,C),x(2,C1),postcomplement].`

The index `postcomplement` selects a subset of the pool of constraints that can apply to $\overline{X}$ rules, some of which are shown in Figure 2, in Prolog-like pseudo code.

```
constraint(+Index,+Head,+Satellite,−MotherNode)

% right compl of preposition
constraint(postcomplement,
                Head,
                Satellite,
                MotherNode) ←

          Head has_category prep,
          Head = MotherNode.

% right complement of adj, n, adv.
constraint(postcomplement,
                Head,
                Satellite,
                MotherNode) ←

          Head is lexical,
          thetamark(Head, Satellite, MotherNode).
```

Figure 2: The constraints indexed to the `postcomplement` rule

Constraints can be divided logically into two groups depending on their function in the parser. Some constraints can affect the way the parser attaches nodes to the already built structure, *i.e.* they can affect the shape of the tree. Other constraints do not: they simply annotate the nodes of a tree already built. Categorial restrictions on the cooccurence of phrases can affect the shape of the phrase marker. On the other hand, morphological feature annotation and feature percolation from the head of the phrase to its maximal projection do not really affect structure-building decisions during the parse. We take advantage of this fact by disassociating feature checking that affects structure-building operations from feature percolation.

On one hand, the separation of rules from constraints has two main advantages. Firstly, it engenders a succinct grammar, because it reduces the number of $\overline{X}$ rules. We obtain the multiplicative effect of several interacting principles, while only using memory resources which correspond to the sum of the sizes of the principles. Moreover, this design achieves language-independence, since it uses highly abstract rules. The same system of constraints, appropriately modified, is used in the Arabic parser as well. For instance, Arabic also has a rich agreement system and the parser uses the same agreement checking mechanism.

On the other hand, since category-neutral rules are used, the parser can generate many structural hypotheses which need to be filtered out. This approach raises the so-called *interleaving* problem, for both Arabic and German: how are the constraints and the phrase structure rules going to be coupled? There are three possible options:

1. All possible phrase structure rules are used, constraints are applied to a forest of trees. This approach is adequate, but it is very space intensive, a forest of hundreds of trees can be built for even small grammars, and it is not very explanatory, in that the entire search space is visited.

[5] Interestingly, an approach where empty categories are postulated as soon as possible, with a partially top-down procedure, has also been used by Crocker (1993). With this design, German and English can be parsed by the same algorithm.

2. All constraints apply at every reduction. This approach is also adequate, but it is not explanatory or efficient, because it applies many constraints in configurations where they are vacuously true.

3. Only a subset of the constraints applies to every rule.

In this system we have adopted the third approach. We have implemented a linking mechanism based on the syntactic configuration. In our system, configurations are partitioned into four main types (following Kornai and Pullum 1990): complement, specifier, adjunct and modifier. Each configuration can appear in two linear orders. A subset of the constraints is indexed to each rule, and must be satisfied for the rule to apply.

While this approach has the problem of not being sufficiently general, since careful tailoring of the interleaving of structure, category and other principles was needed, it is of interest because we found that it eliminates non-determinism more than other approaches. For example, our algorithm to insert empty categories, which relies on structural licensing and theta assignment, is able to eliminate incorrect empty categories as soon as they are postulated, thus it never incurs the explosion, found for example, in principle-based parsers which use functional determination of empty categories (Fong 1991). Also, this approach does not reach the level of specificity that would confine its applicability to only one language. Although we are not able to propose an algorithm to compile automatically the interleaving of principles and rules, we propose a schema that works for such different languages as German and Arabic.

To illustrate, the complete set of $\overline{X}$ rules and the main constraints indexed into each rule are shown in Table 1. Table 2 shows what features are manipulated by each constraint. Finally, the entire pool of constraints is shown in Table 3.

RULE	CONSTRAINT
prespecifier	categorial selection agreement θ-marking feature percolation
postcomplement	categorial selection lexical/functional selection conjunction of likes θ-marking feature percolation
precomplement	categorial selection θ-marking feature percolation
premodifier	categorial selection feature percolation
postmodifier	categorial selection feature percolation
preadjunct	categorial selection feature percolation
postadjunct	categorial selection feature percolation

Table 1: Interleaving of Rules and Constraints

CONSTRAINT	FEATURES
agree	number of head and sister person of head and sister nominative case for sister
categorial selection	category of head and sister
lexical/ functional	category of node
θ-marking	obligatory case and θ-role or optional case and θ-role θ-grid: θ-roles and case
conjunction of likes	category of conjunct
feature percolation	number of head and sister gender of head and sister person of head and sister *wh*-feature

Table 2: Constraints and their Range of Features

5 Error Handling

Since the parser described so far is to be used for tutoring, one very important module of the system is the error handler which detects and diagnoses mistakes. Error tolerance is important to avoid aggravating situations in which the student interacts with a system which is not sufficiently flexible, since the typical user of this system is likely to produce ill-formed input of several kinds: misspellings, agreement errors, (such as wrong declension for nouns and adjectives or wrong conjugation for verbs), syntactic mistakes, (such as putting words in the wrong order, in Ger-

CONSTRAINT	FUNCTION
agree	• checks case assignment and person and number agreement between verb and subject • percolates intersection of features to mother node
categorial selection	• checks that category of head and sister are compatible • percolates category of head
lexical	• checks membership of node to lexical categories
functional	• checks membership of node to functional categories
θ-marking	• checks availability of θ-role for sister • assigns a θ-role • modifies the θ-grid of the head
conjunction of likes	checks categories of conjuncts
percolate	• checks number, gender, and person features • percolates features to mother node

Table 3: The Constraints

man, *e.g.* failing to put the verb at the end of an embedded clause) and also incorrect choice of words semantically, using for instance a movement verb like *legen* instead of the static verb *liegen* to mean *being located.* The parser must also detect and identify the error, while being able to proceed with the analysis. Because error detection is used to build a model of the student and to determine the sequence of learning activities, the diagnostic messages must be accurate, rather than generic. Accuracy is furthermore crucial in choosing the correct default substitute value for a piece of information, which is missing because the input is incorrect. Finally, the system must be flexible, namely detect and diagnose errors at different levels of restrictiveness.

To meet these criteria we have built an error handling facility which is constituted of three logical components. One component performs error detection and produces default values for the mother node if feature matching fails. Upon reduction of a rule, some constraints must be met. One of the tasks of these constraints is to compute the feature set to be assigned to the node that results from the reduction. For example, upon reducing the rule *prespecifier* shown in (17) a constraint on agreement must be satisfied.

(17) `x(2,C) ==>[x(2,C1),x(1,C),prespecifier].`

The constraint states that when reducing a verb and a subject, they must agree in person and number and the subject is assigned nominative case. If agreement succeeds then the result of the unification is used to annotate the mother node, while the error list is empty. If it does not succeed, then the feature to annotate the mother node are determined by default (usually the features of the head are simply copied), and the error list will contain an error code used to diagnose what kinds of errors have occurred.

A second component of the error handler produces messages, based on the code passed by the parser, and retrieves the lexical items which caused the error, by traversing the subtrees in the tree stack. If the error list contains more than one error code then more than one message is generated.

The third component is a set of control switches, which determine how restrictive the parser is going to be in diagnosing errors and reporting them to the tutoring module. Upon detection of a mistake, the error handler checks whether the switch for that particular kind of error is on; if it is, an error message is produced, otherwise the parse proceeds silently.

The use of constraints which are separate from phrase structure rules is crucial in supporting error tolerance. As discussed above, some of the constraints for feature assignment are not needed to determine the shape of the tree. Thus even if such features are missing, the parse tree can be constructed for semantic analysis. At present the error handler in the German parser can detect the following types of errors:

(18)
- Subject verb agreement
- Noun-adjective agreement
- Case errors
- Complement selection errors

- Preposition selection errors
- Errors in selection of auxiliary by verbs
- Word order errors
- Spelling errors

The Arabic error handler uses the same mechanisms to handle a variety of errors, mainly in involving agreement in nominal phrases, which can be classified as follows:

(19)
- Plural formation errors
- Agreement within *noun construct* errors
- Adjective-noun agreement errors
- Subject-verb agreements errors

6 Evaluation

In this section we assess how the initial design goals of robustness, flexibility, portability, and efficiency have been met.

There are two sides to the definition of how robust the system is. First, how wide the linguistic coverage of the natural language processing system is, and second, how tolerant the system is to incorrect input. We have already discussed the error handling ability in some detail.

As far as coverage is concerned, the German parser handles declarative, imperative, and subordinate clauses, *wh*-questions and yes-no questions, topicalizations, inversions, conjunctions, constructions with multiple verbs and with modals.

The Arabic parser handles simple declarative sentences with differing word orders, imperatives, subjectless sentences, *wh*-questions and yes/no questions, simple relative clauses, noun construct constructions, clitic constructions, simple embeddings, and sentences with unexpressed verbs, which are very common in Arabic, and must be distinguished from fragments.

Table 4 shows the percentages of success and failure of the parser on a batch of 205 test sentences for German, which were designed by a team of educators for foreign language training (which did not include any of the authors.) Batch test suites are being constructed for the Arabic parser. The German parser is already supporting prototype lessons. Both the Arabic and the German error sets, which are incorporated in the test sentences, were influenced by an analysis of the needs and by real errors made by foreign language students at the intermediate level. In both parsers there is a good fit between the errors diagnosed, the constructions handled, and the needs imposed by the tutoring application. In the Appendix we exemplify some constructions that the parser can handle.

Flexibility is a different way of looking at the features that support robustness. Our parsing system is flexible in the sense that it is modular and that some modules (the feature annotation constraints) may or may not be incorporated in the parser. For example, we can support two versions of this NLP system: a restrictive and a permissive version. In the latter, feature agreements may be ignored, thus partly ill-formed sentences can still be produced by the student without penalty, while in the former version all the errors are detected.

Thirdly, the portability of the design is very satisfactory. Because we make use of a modular design and a theory of grammar that encodes universal principles, we believe that many parts of this implementation could be used for other languages. Of course, the stored words would be different, but the same design and indeed entire pieces of software could be simply incorporated in the parser for a new language. Merlo (1992) has kept several features of this design in an LR parser for English. The adaptation of the entire system to two very different language, German and Arabic, is done. Work is underway to adapt the system to Spanish.

The system is very compact: the German lexicon contains 5000 roots and the Arabic lexicon contains 500 roots. Since both languages have very productive morphological system, this corresponds to sufficiently large vocabularies. The German parser amounts to 1632 lines of Prolog code, while the Arabic parser to 1736 lines. The system fits on a PC platform with all the software necessary to run the lesson, *i.e.* the tutoring system and the software for the multi-media interface, which includes some very large audio files. The system is able to provide feed-back to

the student at the same speed of a human tutor.[6]

	correct	incorrect	total
input	141.0	64.0	205
parsed	135.0	61.0	196
%	96.4	95.3	95.6

Table 4: Percentage of Successful Parses on Batch Sentences

7 Conclusions

This paper has presented the design and implementation of a parser for German and Arabic, currently used in a tutoring system for computer-aided foreign language training. This is a good application to test the robustness and flexibility of a parsing sytem, since the input is often ill-formed. Moreover, reusability for different languages imposes a portable design. We have illustrated how to adopt a principle-based approach, where linguistic theory is used as directly as possible.

We have discussed the issues related to interaction of principles, recovery of phrase structure using $\overline{X}$ theory, and recovery of long distance dependencies, showing that a principle-based approach provides an interesting experimental answer. We have illustrated the different design choices by several linguistic examples, which cover an interesting range of constructions in German and Arabic.

Acknowledgements

This work was supported by research grant No. 920427-7519 "New Cognitive Technologies to Improve Foreign Language Training" from the Army Research Institute to the University of Maryland and MicroAnalysis and Design, and also by grant No. 15-890071-62 "Applications of AI to Foreign Language Training" to the University of Maryland and SAIC. Thanks also to the Academic Society of the University of Geneva.

[6]Preliminary measures on the two parsers for an uncompiled (interpreted) version on a SUN workstation have given an average speed of 14 words/second for German and 27 words/second for Arabic.

References

Aho, A.V. — J.D. Ullman (1972) *The Theory of Parsing, Translation and Compiling.* Englewood Cliffs, NJ: Prentice-Hall.

Azadegan, S. — J. Martin — P. Merlo — A. Weinberg (forthcoming) *A Government-Binding Parser for Foreign Language Training.* UMIACS-TR, College Park, MD: UMCP.

Berwick, R. (1991) "Principle-Based Parsing". In: Sells P. & S.M. Shieber & T. Wasow (Eds.): *Foundational Issues in Natural Language Processing.* 115–226. Cambridge, MA: MIT Press.

Berwick, R. — A. Weinberg (1984) *The Grammatical Basis of Linguistic Performance.* Cambridge, MA: MIT Press.

Chomsky, N. (1965) *Aspects of the Theory of Syntax.* Cambridge, MA: MIT Press.

Chomsky, N. (1970) "Remarks on Nominalization". In: Jacobs R. & P. Rosenbaum (Eds.): *Readings in Transformational Grammar.* 184–221. Waltham, MA: Ginn & Co.

Chomsky, N. (1973) "Conditions on Transformations". In: Anderson S. & P. Kiparsky (Eds.): *A Festschrift for Morris Halle.* 232–286. New York, NY: Holt, Reinhart and Winston.

Chomsky, N. (1977) "On *Wh*-movement". In: Culicover P. & T. Wasow & A. Akmajian (Eds.): *Formal Syntax.* 71–132. New York, NY: Academic Press.

Chomsky, N. (1981) *Lectures on Government and Binding.* Dordrecht: Foris.

Crocker, M.W. (1993) "Properties of the Principle-Based Sentence Processor". In: *Proceedings of the 15th Annual Cognitive Science Society.* Boulder, CO.

Dorr, B.J. (1987) *UNITRAN: a Principle-based Approach to Machine Translation.* Ms Thesis, MIT. Cambridge, MA.

Dorr, B.J. (1993) *Machine Translation: A View from the Lexicon* . Cambridge, MA: MIT Press.

Fong, S. (1991) *Computational Properties of Principle-based Grammatical Theories.* Ph.D. Dissertation, MIT. Cambridge, MA.

Gazdar G. — E.Klein — G.Pullum — I.Sag (1985) *Generalized Phrase Structure Grammar.* Oxford: Blackwell.

Jackendoff, R. (1977) *X' Syntax: A Study of Phrase Structure.* Cambridge, MA: MIT Press.

Kayne, R. (1984) *Connectedness and Binary Branching.* Dordrecht: Foris.

Kornai, A. — G. Pullum (1990) "The X-bar Theory of Phrase Structure". In: *Language* 66, 24 – 50.

Merlo, P. (1992) *On Modularity and Compilation in a Government and Binding Parser.* Ph.D. Dissertation, University of Maryland at College Park. College Park, MD.

Shieber, S. (1986) *An Introduction to Unification-Based Approaches to Grammar.* Chicago, IL: University of Chicago Press.

Thiersch, C. (1978) *Topics in German Syntax,* Ph.D. Dissertation, MIT. Cambridge, MA.

Tomita, M. (1985) *Efficient Parsing for Natural Language.* Hingham, MA: Kluwer.

Wanner, E. — M. Maratsos (1978) "An ATN Approach to Comprehension" In: Halle M. & J. Bresnan & G.A. Miller (Eds.): *Linguistic Theory and Psychological Reality.* 119–161. Cambridge, MA: MIT Press.

Appendix

Simple Locative

(1) *Ihr liegt im Sueden*
you.pl lie in the.dat South
'You are in the South'

(2) *Der Berg liegt Suedlich der*
the mountain lies south the.gen
Stadt Lauterbach in Hessen
city Lauterbach in Hessen
'The mountain is south of the city of Lauterbach in Essen'

(3) *Yuujadu maa un fii l-mityaahati*
there be a water in the Mityaha
'There is water in Al-Mityaha'

Simple Predicative

(4) *Ihr frau ist klug*
your.pl wife is smart
'Your wife is smart'

(5) *Al-Mityaahatu laysat 9alaa Tariiqin*
Al-Mityaahatu not be on the road
'Al-Mityaha is not on a road'

Simple Transitive

(6) *Ich habe die Antwort gefunden*
I have the answer found
'I have found the answer'

(7) *Man hat einen guten Rundblick in*
one has a good view in
das Tal hinunter
the valley below
'One can have a good view of the valley below'

Simple Intransitive

(8) *Links neben dem Bach befindet*
left near the.dat stream finds
sich die Eisenbahnlinie
itself the railway-track
'To the left of the stream finds itself (lies) the railway-track'

Conjunctive Phrases

(9) *Ich blicke nach links, nach*
I look toward left toward
Norden und nach Osten
north and toward east
'I look to the left, to the North and to the East'

(10) *Sahraawiyyatin wa Hajariyyatun*
desertlike and stony
'desertlike and stony'

Simple Questions

(11) *Wo stehen wir?*
where stand we
'Where are we standing?'

(12) *In welche Richtung waechst die*
in which direction grows the
moderne Stadt Lauterbach?
modern city Lauterbach
'In which direction is the modern city of Lauterbach?'

(13) *Maa huwa 9adaddu T-Turuqi*
What he number the roads
l-mawjuudaati fii d-dibbati
the found in Al Dibdibba
'How many road are found in Al-Dibdibba?'

Imperatives

(14) *Beschreiben Sie die*
describe.pl you.formal the
Umgebung der Stadt
sorroundings the.gen city
'Describe the sorroundings of the city'

(15) *Sifa l-buyuuta fii d-dibdibbati*
Describe the houses in the AlDibdibba
'Describe the houses in Al-Dibdibba'

(16) *da9naa nufakkiru bi mandharin*
we will consider conj land
Tabii9iyyin namuuthajiyyin fii gharbi
nature typical in west
l-bilaadi
the country
'Let's consider a typical landscape in the West of the country'

Modals

(17) *Was kann man auf dieser Skizze*
what can one on these sketches
von Lauterbach erkennen?
of Lauterbach recognize

'What can one recognise in these sketches of Lauterbach?'

(18) *Man kann ein Burg erkennen*
one can a fort recognise

'One can recognise a fort'

Inversion

(19) *Oestlich von Lauterbach liegt*
east of Lauterbach lies
Angersbach
Angersbach
'East of Lauterbach lies Angersbach'

Embedded Constructions

(20) *Ich denke, daß Peter und Hans*
I think that Peter and Hans
nach Deutschland gegangen sind
towards Germany gone are
'I think that Peter and Hans have gone to Germany'

An Algorithm for the Construction of Dependency Trees

Gerrit F. van der Hoeven

University of Twente, Department of Computer Science, Section SETI
P.O. Box 217, 7500 AE Enschede, The Netherlands
email: vdhoeven@cs.utwente.nl

Abstract

A casting system is a dictionary which contains information about words, and relations that can exist between words in sentences. A casting system allows the construction of dependency trees for sentences. They are trees which have words in roles at their nodes, and arcs which correspond to dependency relations. The trees are related to dependency trees in classical dependency syntax, but they are not the same. Formally, casting systems define a family of languages which is a proper subset of the contextfree languages. It is richer than the family of regular languages however. The interest in casting systems arose from an experiment in which it was investigated whether a dictionary of words and word-relations created by a group of experts on the basis of the analysis of a corpus of titles of scientific publications, would suffice to automatically produce reasonable but maybe superficial syntactical analyses of such titles. The results of the experiment were encouraging, but not clear enough to draw firm conclusions. A technical question which arose during the experiment, concerns the choice of a proper algorithm to construct the forest of dependency trees for a given sentence. It turns out that Earley's well-known algorithm for the parsing of contextfree languages can be adapted to construct dependency trees on the basis of a casting system. The adaptation is of cubic complexity. In fact one can show that contextfree grammars and dictionaries of words and word-relations like casting systems, both belong to a more general family of systems, which associate trees with sequences of tokens. Earley's algorithm cannot just be adapted to work for casting systems, but it can be generalized to work for the entire large family.

1 Associating trees with sentences

This paper is about formal systems which associate trees with sequences of symbols. Most of the contents of the paper deal with definitions, the formal properties of the systems defined, common generalizations of new and well-known systems, and finally parsing problems. First however, we will describe an experiment which gave rise to the formalisms we introduce here. The experiment is as follows.

A group of experts is given a set of titles of scientific publications in their field of expertise. As a first step, they are asked to give a structural analysis of the titles. More precisely: their task is to draw lines between related words in each of the titles of the corpus, in such a way that every title gets a tree structure.

The words of the title are the nodes of the tree, the arrows connecting mothers and daughters in the tree stand for: 'in some sense related'. Such an analysis applied to the title of this paper might yield a tree like the one in figure 1. There is one restriction concerning word order the experts must obey in drawing their trees. The restriction is, that if they relate word b to word a, then no word c which is at the other side of a than b is in the textual order, can be related to b.

The second step is, to ask the experts to motivate their tree constructions. The motivation must take a specific form. They are asked to give a name to the lines connecting mother- and daughter-words in their trees, in a consistent way for all titles. In this way they are supposed to make explicit which relations between words they

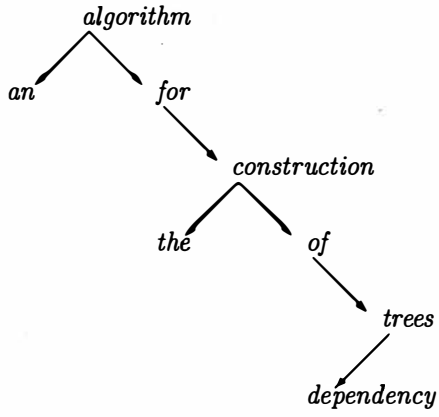


Figure 1: A tree for the title

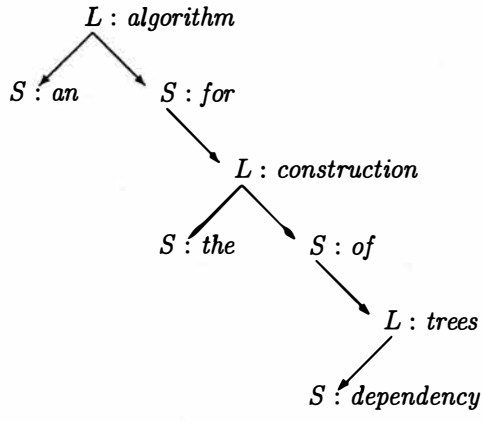


Figure 2: An attributed tree for the title

consider important. Moreover, they are asked to name the characteristics of the individual words in every tree. Thus it is made explicit what the properties of the individual words are that make them fit in a particular relationship to one another.

The final step would be, to redraw the trees in such a way that the relation names assigned to lines connecting mothers and daughters, are now assigned to the daughters, together with the characteristics of the daughter words.

From this second representation, the original one can be easily reconstructed, since there is always only one line in the tree to which the relation component in the dressing of a daughter can belong. This final tree translation does not affect the structure of the trees nor does it contribute to the insight into their structure. It is relevant for technical purposes: now we have trees in which only the nodes have attributes, instead of both nodes and arcs.

A schematic representation of a final result tree (with just two simple attributes L and S, where a real tree would have more and more complex ones) is the tree in figure 2.

The outcome of such an experiment could be interesting for all sorts of reasons. Our interest is simply to use the trees, the word characteristics and the relations between words as indicated by the experts, to construct similar trees for titles of publications that the experts did not consider.

The basic idea for extrapolation of the results of the experiment is to abstract from the trees that are delivered, and to concentrate on *word profiles* that can be derived from the trees. A word profile is roughly a triple consisting of an attribute, and two sets of attributes. A profile for a word can be derived from a set of trees by first collecting all trees in which the word has the same attribute. Next the set of attributes assigned to daughters of that word in any of the trees are collected. Finally this set of daughter attributres is split in two, possibly overlapping, subsets. One has the attributes assigned to daughters which occur to the left of the given word, and the other has the attributes assigned to words which occur to the right. Note that a word can occur with different attributes, and that therefore a word can have more than one profile. The formal notion of a casting system introduced below, gives the precise elaboration of this idea.

Before we turn to the formal definitions and their properties, a few remarks are in order. The first remark concerns the experiment described above. It was never properly conducted. There have been experts drawing trees for titles of scientific publications, but they were the same group as the ones who used the resulting trees for the analysis of new titles. Although it was not known beforehand to which new titles the analysis method would be applied, the experts making the original analyses were clearly aware of the ways in which their results would be used. Their discussions therefore concentrated not so much on the actual analyses they made, but more on the generality of the relations between words and the characteristics of individual words they introduced. Moreover, the corpus of titles they considered was too small to draw any firm conclusions from the outcome of the experiment anyway. But the results were not discouraging.

The second remark concerns the kind of trees we consider and the notion of word profile. In shape, the trees are very much like dependency trees. What we ask the experts to do, could rightly be called dependency analysis. The syntactical claims in our approach however, are far from classical dependency syntax. In fact, we will present a system that is capable of assigning trees to well-formed utterances, but that will assign trees just as easily to many ill-formed utterances. The question what makes a sentence or phrase correct, let alone the explanation of correctness at any level of adequacy, does not interest us. What we want, is to have a tree shaped representation of an utterance which organizes the information in that utterance in a way that is both manageable and acceptable to a human reader or hearer of the utterance.

As for the word profiles, if one thinks of the attributes for words as semantic categories, and omits the left-of/right-of distinction, a word profile bears some resemblance to a case frame. In fact, it seems that the analysis we consider here could just as well be performed on the basis of a dictionary of case frames, as on the basis of a dictionary of word profiles that are derived from a corpus of handmade analyses.

1.1 Casting systems and dependency trees

A casting system is nothing but the formal description of a dictionary of word profiles, as introduced informally above. There is a slight change of terminology however. What we called 'words' above, are 'actors' in the formal representation, and what we called 'attributes', are now 'roles'. A casting system tells which actors can play which roles, and what supporting roles the actors in their roles expect to their left and to their right.

Strictly formal, a casting system is a seven tuple of sets, symbols and relations. It fixes a relation between sequences of 'actors' and dependency trees. It is a dictionary of words, word roles, and co-occurrence relations between words and roles.

Definition 1 A *casting system* Γ is a seven tuple with the following components:

- A, the *actor set* of Γ. A is a finite alphabet. Its elements are actors.
- P, the *set of roles* of Γ. P is a finite set.
- L, the set of *leading roles* of Γ. L is a subset of P.
- ι, the *invisible role* of Γ. ι is a distinguished element of P.
- $\Box : \Box$, the *can-be-played-by relation* of Γ. It relates roles and actors. If p is a role and

Actors	{ *an*, *algorithm*, *for*, *the*, *construction*, *dependency*, ... }
Roles	$\{L, S, \iota\}$
Leading roles	$\{L\}$
The invisible role	ι
Can-be-played-by	L : *algorithm*, L : *construction*, ...
	S : *an*, S : *for*, S : *the*, S : *dependency*, ...
Can-be-combined-left	$S\backslash L$: *algorithm*, $S\backslash L$: *construction*, ...
Can-be-combined-right	L : *algorithm* $/S$, S : *for* $/L$, ...
Combine-with-ι	S : *an* $/\iota$, S : *dependency* $/\iota$, ...
	$\iota\backslash S$: *an*, $\iota\backslash S$: *for*

Figure 3: A simple casting system

a is an actor then we write $p : a$ to express that p can be played by a.

- $\square\backslash\square : \square$, the *can-be-combined-left relation* of Γ. It relates roles with actors and roles. If p and q are roles, and a is an actor, then we write $q\backslash p : a$ to express that a in role p can play together with any actor in role q to its left.

- $\square : \square/\square$, the *can-be-combined-right relation* of Γ. The counterpart of the previous relation in the following sense: we write $p : a/q$ to express that a in role p can play together with any actor in role q to its right.

It should be obvious that the can-be-played-by, can-be-combined-left and can-be-combined-right relations give us the ingredients of a word profile. The special status of the set of leading roles is, that it contains the roles that can appear at the root of a well-formed tree. The invisible role is important in the 'combine' relations. Possibility of combination with the invisible role indicates that an actor can occur without support of other roles, i.e. without daughters in a dependency tree.

An example of a small casting system, corresponding to the dependency tree shown earlier for the title of this paper, is in figure 3.

A casting system is just the rules of the game. The rules can be derived from a given set of trees. But the game is the inverse: to associate trees with sequences of actors. That is what the following definition is about.

Definition 2 Let Γ be a casting system with actor set A, and let u be a string of actors.

A *casting tree* or a *dependency tree* for u w.r.t. Γ is a directed graph T. The nodes of T are pairs (p, α), with p a role of Γ, and α an occurrence of an actor of Γ in u. The graph T has the following properties:

- it is a tree,
- the role of every node can be played by the actor of the node;
- if (q, β) is a successor of (p, α) and the occurrence β is to the left of the occurrence α, then $q\backslash p : a$, where a is the actor of which α is an occurrence; if β is to the right of α, then $p : a/q$;
- if node (p, α) has no successors (q, β) with β to the left of α, then $\iota\backslash p : a$; if there are no successors (q, β) with β to the right of α, then $p : a/\iota$;
- with every node there is a segment v of u, which consists of the actors in the node and in its descendants. In particular, the root node corresponds to the entire sequence u.

A casting system has an associated formal language. The existence of a dependency tree w.r.t. the casting system determines whether or not a string of actors belongs to the formal language.

Definition 3 Let Γ be a casting system, with actor set A. The *language associated with* Γ is the set of actor sequences in A^* which have a dependency tree w.r.t. Γ. We call this language $\mathcal{L}_\Gamma$.

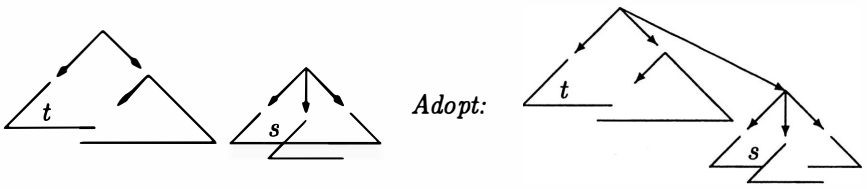


Figure 4: Where t adopts s

The family of casting languages has some peculiar properties. We mention the following facts without proof.

Fact 1 If Γ is a casting system, $\mathcal{L}_\Gamma$ is the associated language, a and b are actors, and ab is a string in $\mathcal{L}_\Gamma$, then ab^n or $a^n b$ is in $\mathcal{L}_\Gamma$ for every $n > 0$. More general, in every string in $\mathcal{L}_\Gamma$ with two or more actors, there is at least one actor which can be repeated arbitrarily often, and the resulting string will again be in $\mathcal{L}_\Gamma$.

Fact 2 If Γ is a casting system, and $\mathcal{L}_\Gamma$ is the associated language, then $\mathcal{L}_\Gamma$ is contextfree.

Fact 3 There are regular languages $\mathcal{L}$, for which no casting system Γ exists such that $\mathcal{L} = \mathcal{L}_\Gamma$.

Fact 4 There are casting systems Γ which have an associated language $\mathcal{L}_\Gamma$ which is not regular.

Fact 1 shows that casting systems are not the proper systems to distinguish between ill- and well-formed phrases of a natural language. Fact 3 is an immediate consequence of fact 1. From facts 2 to 4 we see that casting languages do not have a proper place in the Chomsky hierarchy, but are 'somewhere in between regular and contextfree'.

There are quite a number of open problems concerning casting systems. E.g. is it decidable whether two casting systems have the same associated language or not?

We will not go into formal properties of casting systems here, nor will we further pursue the question of their suitability for the description or the processing of natural language. The second half of this paper deals with the parsing problem casting systems pose, and the solution to that problem which is found in a common generalization of casting systems and contextfree grammars.

2 Parsing: the construction of a dependency tree for a given sentence

In this section we discuss the problem of constructing dependency trees for sentences on the basis of a casting system Γ. But what we shall do is *not* to present a parsing algorithm for casting languages. Our approach is to consider the association of analysis trees with sequences of symbols in general terms, independent of whether the associated trees are dependency trees or e.g. contextfree parse trees. First we will show that such a general approach, of which dependency trees and contextfree parse trees are both an instance, indeed exists. Then we consider the parsing problem for the generalized notion of tree association. We conclude that the generalized notion has such characteristics that it allows an Earley-like parsing strategy. It follows that dependency trees can be constructed by an Earley-like algorithm. To obtain the actual Earley algorithm for contextfree languages from the generalized version, optimizations are needed which are typical for the contextfree case. We shall not go into these optimizations.

The kernel of the generalized notion of tree association, is the notion of FB-system. Strictly formal, an FB-system is a seven tuple of sets and relations. It fixes a set of colored trees, and a relation between colored trees and sequences of symbols.

A *colored tree* is a tree with a mapping from its nodes into a set of colors. The set of colors is just an arbitrary finite set.

In the sequel we shall work with three basic tree forming operations. They are: *Single*, *Adopt*, and *Recolor*, and they are defined as follows:

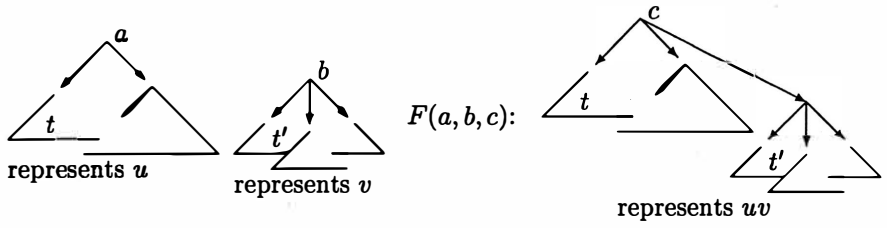


Figure 5: Forward adoption

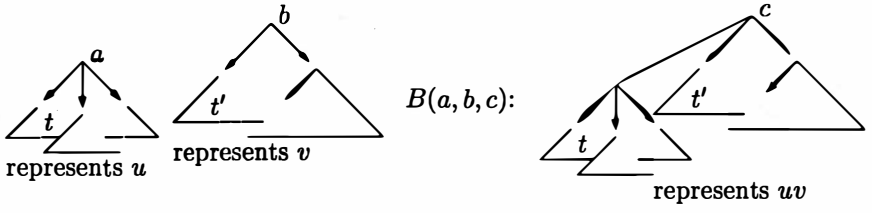


Figure 6: Backward adoption

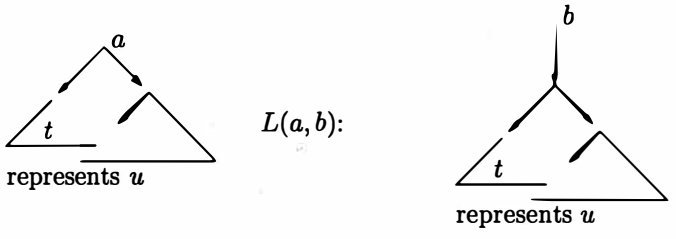


Figure 7: Lift

Single takes a color, and yields a tree consisting of just a single node, which has this color.

Adopt takes two trees, which it turns into one by making the root of the second tree a daughter of the root of the first one (cf. Figure 4).

Recolor finally, takes a tree and a color and 'changes' the root of the given tree to have the given color.

We shall use the phrase: the *color of a tree* to mean the color of the root of the tree. We shall denote this color of t by $\gamma(t)$.

With these preliminaries we are now able to give a precise definition of an FB-system, and to give an interpretation to this formal definition.

Definition 4 An *FB-system* is a seven tuple Φ with the following components:

- C, the *alphabet of colors* of Φ.
- C_R, subset of C with the *admissible root colors*.
- S, the *alphabet of symbols* of Φ. S and C are disjoint.
- R, the *representation relation* of Φ, a relation on $C \times S$. $R(c, s)$ indicates that the color c 'represents' the symbol s. A color which represents a symbol is a *terminal color*.
- F, the *forward relation* of Φ, a relation on $C \times C \times C$. $F(c_0, c_1, c)$ indicates that a tree with color c_0 can adopt one with color c_1, to form a new tree of which the root is (re)colored by c. If c is terminal, then so is c_0. If both are terminal, then they represent the same symbols.

- B, the *backward relation* of Φ, a relation on $S \times S \times S$. $B(c_0, c_1, c)$ indicates the same as $F(c_0, c_1, c)$, except that now c_1 adopts c_0. If c is terminal, then so is c_1. If both are terminal, then they represent the same symbols.
- L, the *lift relation* of Φ, a relation on $S \times S$. $L(c_0, c_1)$ indicates that a tree with color c_0 can be adopted by a single node tree with color c_1. c_1 can not be a terminal color.

FB-systems are named after their characteristic forward- and backward-relations. An FB-system is just a formalism which fixes the rules of the game. The game is to build colored trees, and to associate such trees with sequences of symbols. The next definition (with pictures) tells us how to interpret the contents of an FB-system.

Definition 5 The set of *admissible trees* T_Φ over an FB-system Φ is a set of colored trees. Every tree in T_Φ is an *analysis tree* for a sequence of symbols.

The set T_Φ and the analysis tree relation are inductively defined by the following four clauses:

1. If s is a symbol and the color c represents s, then $Single(c)$ is an admissible tree, it is an analysis tree for s.
2. If t with $\gamma(t) = a$ is an analysis tree for u, and t' with $\gamma(t') = b$ is an analysis tree for v, and $F(a, b, c)$ holds, then $Recolor(Adopt(t, t'), c)$ is an admissible tree, it is an analysis tree for uv (forward adoption, cf. figure 5).
3. If t with $\gamma(t) = a$ is an analysis tree for u, and t' with $\gamma(t') = b$ is an analysis tree for v, and $B(a, b, c)$ holds, then $Recolor(Adopt(t', t), c)$ is an admissible tree, it is an analysis tree for uv (backward adoption, cf. figure 6).
4. If t with $\gamma(t) = a$ is an analysis tree for u, and $L(a, b)$ holds, then $Adopt(Single(b), t)$ is an admissible tree, it is also an analysis tree for u (lift, cf. figure 7).

2.1 A contextfree grammar as an FB-system

To illustrate the concept of FB-system and the association of analysis trees with sequences of symbols, we will present a simple contextfree grammar as an FB-system, and show how the parse tree of a simple sentence can be obtained as an analysis tree according to the foregoing definition.

Terminals	$\{d, n, p\}$
Non-terminals	$\{ NP, PP \}$
Start symbol	NP
Rules	$NP \rightarrow d\ n$, $NP \rightarrow NP\ PP$, $PP \rightarrow p\ NP$
Colors	$\{d \rightarrow d, n \rightarrow n, p \rightarrow p, NP \rightarrow d\ n, NP \rightarrow NP\ PP, PP \rightarrow p\ NP, NP \rightarrow d, NP \rightarrow NP\ PP \rightarrow p\}$
Root colors	$\{ NP \rightarrow d\ n, NP \rightarrow NP\ PP \}$
Symbols	$\{d, n, p\}$
Represent	$R(d \rightarrow d, d), R(n \rightarrow n, n), R(p \rightarrow p, p)$
Forward	(F1) $F(NP \rightarrow d, n \rightarrow n, NP \rightarrow d\ n)$
	(F2) $F(NP \rightarrow NP, PP \rightarrow p\ NP, NP \rightarrow NP\ PP)$
	(F3) $F(PP \rightarrow p, NP \rightarrow d\ n, PP \rightarrow p\ NP)$
	(F4) $F(PP \rightarrow p, NP \rightarrow NP\ PP, PP \rightarrow p\ NP)$
Backward	empty
Lift	(L1) $L(NP \rightarrow d\ n, NP \rightarrow NP)$ (L2) $L(NP \rightarrow NP\ PP, NP \rightarrow NP)$
	(L3) $L(d \rightarrow d, NP \rightarrow d)$ (L4) $L(p \rightarrow p, PP \rightarrow p)$

Figure 8: The example grammar and the corresponding FB-system

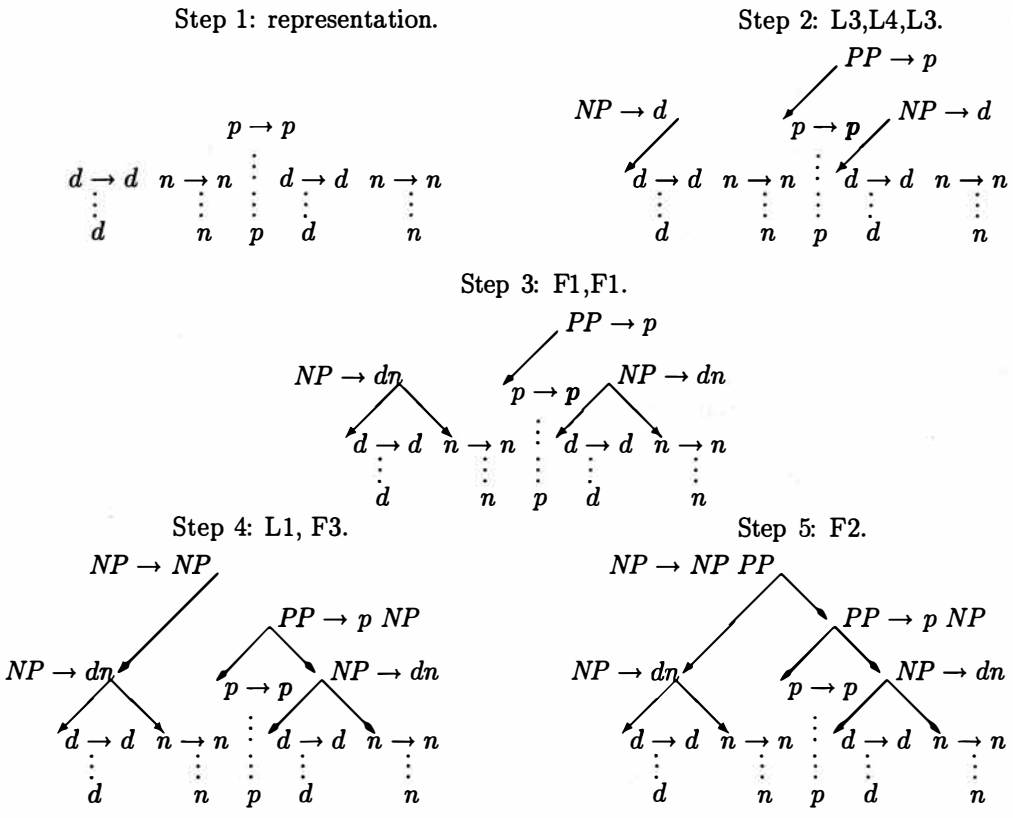


Figure 9: The construction of an analysis tree for *dnpdn*

The simple grammar we consider is shown in the top half of figure 8. The corresponding FB-system is shown in the bottom half of the same figure.

Its set of colors contains an element $x \to x$, for every terminal x of the grammar. It contains also all production rules of the grammar. Finally it contains the 'partial' rules $x \to u$, which are such that $x \to uv$ is a rule of the grammar (both u and v not empty).

The set of root colors of the FB-system contains the colors that are production rules for the start symbol.

Its symbols are the terminals of the grammar.

The representation relation R contains three pairs, one for every symbol.

The idea behind the forward relation F and the lift relation L of the system is, that an analysis tree will always have a color indicating a partially recognized production. The F rules state how to extend partial recognition, the L rules tell how a terminal or a completely recognized non-terminal can be the leftmost symbol in a partially recognized other non-terminal.

The five pictures in figure 9 show the step-by-step construction of an analysis tree for the sequence *dnpdn*. The analysis tree is a parse tree. The steps are:

1. single node trees representing the symbols,
2. three admissible lifts (L3,L4,L3),
3. two admissible forward adoptions (twice F1),
4. an admissible lift (L1), and a forward adoption (F3),
5. an admissible forward adoption (F2).

Actors	$\{d, n, p\}$
Roles	$\{D, N, P, \iota\}$
Leading roles	$\{N\}$
The invisible role	ι
Can-be-played-by	$D : d$, $N : n$, $P : p$
Can-be-combined-left	$D\backslash N : n$, $\iota\backslash D : d$, $\iota\backslash P : p$
Can-be-combined-right	$D : d/\iota$, $N : n/P$, $N : n/\iota$, $P : p/N$
Colors	$\{D : d, \bullet N : n, N : n, P : p\bullet, P : p\}$
Root colors	$\{N : n\}$
Symbols	$\{d, n, p\}$
Represent	$R(D : d, d), R(\bullet N : n, n), R(P : p\bullet, p)$
Forward	(F1) $F(\bullet N : n, P : p, \bullet N : n)$,
	(F2) $F(N : n, P : p, N : n)$,
	(F3) $F(P : p\bullet, N : n, P : p)$,
	(F4) $F(P : p, N : n, P : p)$
Backward	(B1) $B(D : d, \bullet N : n, N : n)$,
	(B2) $B(D : d, N : n, N : n)$
Lift	empty

Figure 10: The example casting- and the corresponding FB-system

2.2 A casting system as an FB-system

In a similar way as the contextfree grammar above, we can present a casting system as an FB-system, and show how the analysis tree construction yields a dependency tree. We take the casting system of the top half of figure 10 as an example.

The corresponding FB-system is in the second half of the same figure.

Its set of colors contains all possible role–actor pairs according to the can-be-played-by relation. Role–actor combinations which cannot do without support, i.e. which cannot be combined with the invisible role, also appear 'dotted' in the color set. The dot marks the side at which support is obligatory.

There is one admissible root color, corresponding to the leading role.

The symbols are, as before, the actors of the casting system.

The representation relation R contains three pairs, one for every symbol.

F has four triples, B has two, and L is the empty relation, no color can be lifted to another.

Note that F and B correspond to the can-be-combined relations. Note also that a dot disappears in recoloring whenever a dotted color adopts a color at the side of the dot.

Note finally that the absence of Lift here and the importance of Lift in the case of contextfree grammars reflect the fact that every node in a dependency tree represents a symbol, whereas in parse trees the internal nodes are representatives of constituents.

The four pictures in figure 11 show the step-by-step construction of an analysis tree for the sequence *dnpdn*. The resulting analysis tree is a dependency tree for the sequence. The steps in its construction are:

1. single node trees, representing the individual symbols,
2. two admissible backward adoptions (twice B1),
3. a forward adoption (F3),
4. a forward adoption (F2).

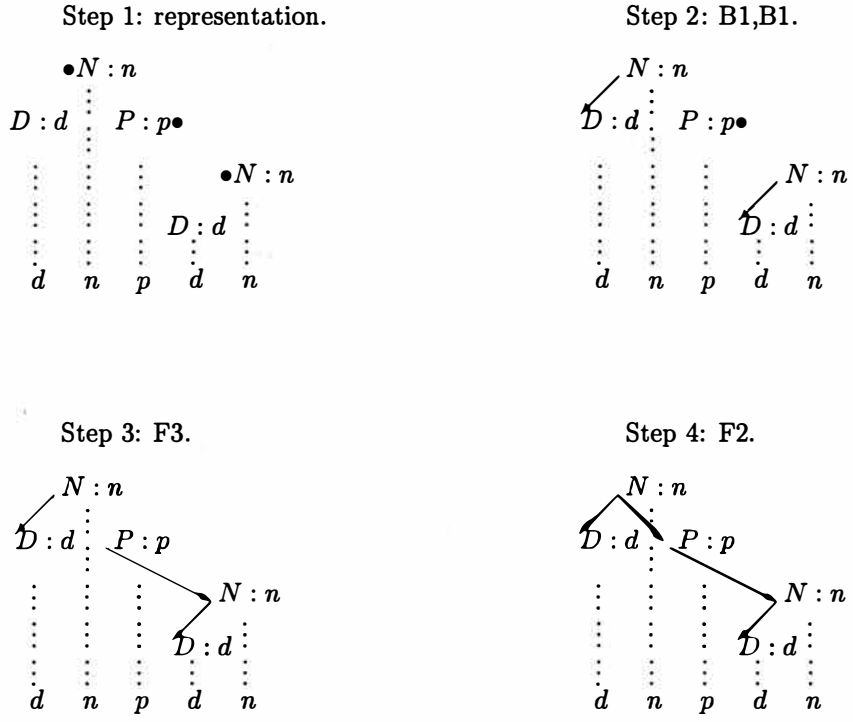


Figure 11: The construction of an analysis tree for *dnpdn*

2.3 Recognizing sequences which have an analysis tree

The two examples are of course not a proof of the fact that every contextfree grammar and every casting system can be represented as an FB-system. Such a proof can be given, it is in fact not difficult. But it is tedious, and we shall not present it here. Hopefully, the examples are enough to suggest the general techniques applicable for translating the one formalism into the other.

The goal of the introduction of FB-systems was to come to a uniform approach to parsing. It is parsing we shall now concentrate on. That is to say, the actual problem of parsing is to construct an analysis tree, if any, or even better, all analysis trees for a given sequence of symbols w.r.t. a given FB-system. What we present here is a strategy for recognition. Strictly speaking, the algorithm we present (not in full algorithmic detail), is capable only of deciding whether a given sequence has an associated analysis tree or not, and it does not produce the trees. But it is a well-known technique, and a minor adaptation to the algorithm, to keep track of the ways in which items are combined during recognition. Such additional information is sufficient to produce all analysis trees.

The central notion in the recognition algorithm is the notion of an item.

Definition 6 An *item* for a given FB-system Φ is an element of the Cartesian product $N \times C$, in which N is the set of positive natural numbers and C is the color set of Φ.

The overall structure of the recognizer is as follows.

The recognizer R w.r.t. Φ is an algorithm which expects an input sequence $(a_1, \ldots, a_n)$ which consists of symbols a_i of Φ.

As a result it produces a sequence of item sets $(I_0, \ldots, I_n)$.

I_0 is empty.

Every next I_{k+1} is computed from the symbol a_{k+1} and the preceding initial segment $(I_0, \ldots, I_k)$ of the result sequence.

The computation of I_{k+1} will yield a set of items (m, c) in which $m \leq k+1$.

The interpretation of $(m, c) \in I_k$ is: there is an analysis tree t for the segment $(a_m, \ldots, a_k)$, with $\gamma(t) = c$.

Recognition is expressed as: the last item set in the sequence produced, i.e. I_n, contains an item $(1, c)$ in which c is an admissible root color of Φ.

3mm

To be precise: the construction of the next item set I_{k+1} from the previous ones $(I_0, \ldots, I_k)$ and the next symbol a_{k+1} proceeds as follows:

It starts with the set

$K = \{(k+1, c) | R(c, a_{k+1})\}$

Of this set, the completion is constructed. The completion of K is the smallest set J satisfying:

- $K \subseteq J$
- If $(j+1, d) \in J$ and $(i, c) \in I_j$ and $F(c, d, c')$, then $(i, c') \in J$
- If $(j+1, c) \in J$ and $(i, d) \in I_j$ and $B(d, c, c')$, then $(i, c') \in J$
- If $(j+1, c) \in J$ and $L(c, c')$ then $(j+1, c') \in J$.

This completion is I_{k+1}.

For the complexity of the construction of the completion, the following is relevant:

every set I_j has a number of elements bounded by $C \times j$, where C is the number of colors,

to construct I_{k+1}, all sets I_j, $j < k+1$, must be traversed, every item in these previous sets must be matched against at most C items already in I_{k+1}, and for every item newly constructed at most C lifts must be added.

It follows that the number of steps in the construction of I_{k+1} is bounded by C^2k^2.

The recognition algorithm is of cubic complexity.

3 Conclusions

The creation of dependency trees for utterances on the basis of a dictionary of word profiles, i.e. a casting system, derived from the handmade analysis of a restricted set of utterances, is an interesting approach to structural analysis. To assess the full merits of the approach, further research is necessary however.

The parsing problem for the construction for dependency trees is in many respects the same as that for contextfree derivation trees. In fact, the general notion of FB-system seems to cover all methods of associating trees with sequences which are local, i.e. all methods where the well-formedness of the associated tree is determined by restrictions on the structure of the nodes, and not on the tree as a whole. An Earley-like algorithm of cubic complexity applies to every association of trees to sequences on the basis of such a general FB-system.

References

[1] Earley, J. (1970), "An efficient contextfree parsing algorithm". *Communications of the ACM* 13, 90–102.

[2] Filmore, C. (1968), "The case for case". In: Bach, E. & R. Harms, (Eds.): *Universals in Linguistic Theory*, 1–68. New York : Holt, Reinhart and Winston.

[3] Hoeven, G.F. van der (1992), "An experiment in the syntactical analysis of English noun phrases". *Memoranda Informatica* 92-24, 34pp. Enschede, The Netherlands : University of Twente, Department of Computer Science.

[4] Hoeven, G.F. van der (1992), "An algorithm for the construction of dependency trees", *Memoranda Informatica* 92-39, 29pp. Enschede, The Netherlands : University of Twente, Department of Computer Science.

[5] Hopcroft, J.E. – J.D. Ullman (1979), *Introduction to Automata Theory, Languages and Computation.* Reading, Massachusetts: Addison-Wesley.

[6] Schubert, K. (1987), *Metataxis—Contrastive Dependency Syntax for Machine Translation.* Dordrecht, Providence R.I. : Foris Publications

Integration of Morphological and Syntactic Analysis Based on LR Parsing Algorithm

Tanaka Hozumi, Tokunaga Takenobu And Aizawa Michio

Department of Computer Science, Tokyo Institute of Technology
2-12-1 Ôokayama Meguro Tokyo 152 Japan
email: take@cs.titech.ac.jp

Abstract

Morphological analysis of Japanese is very different from that of English, because no spaces are placed between words. The analysis includes segmentation of words. However, ambiguities in segmentation is not always resolved only with morphological information. This paper proposes a method to integrate the morphological and syntactic analysis based on LR parsing algorithm. An LR table derived from grammar rules is modified on the basis of connectabilities between two adjacent words. The modified LR table reflects both the morphological and syntactic constraints. Using the LR table, efficient morphological and syntactic analysis is available.

1 Introduction

Morphological analysis of Japanese is very different from that of English, because no spaces are placed between words. This is also the case in many Asian languages such as Korean, Chinese, Thai and so forth. In the Indo-European family, some languages such as German have the same phenomena in forming complex noun phrases. Processing such languages requires the identification of the boundaries of words in the first place. This process is often called *segmentation* which is one of the most important tasks of morphological analysis for these languages.

Segmentation is a very important process, since the wrong segmentation causes fatal errors in the later stages such as syntactic, semantic and contextual analysis. However, correct segmentation is not always possible only with morphological information. Syntactic, semantic and contextual information may help resolve the ambiguities in segmentation.

Over the past few decades a number of studies have been made on the morphological and syntactic analysis of Japanese. They can be classified into the following three approaches:

Cascade: Separate the morphological and syntactic analysis and execute them in a cascade manner. The morphological and syntactic constraints are represented separately.

Interleave: Separate the morphological and syntactic analysis and execute them interleavingly. The morphological and syntactic constraints are represented separately.

Single Framework: Represent both the morphological and syntactic constraints in a single framework such as context free grammars (CFGs) and make no distinction between the two analysis.

Representing the morphological and syntactical constraints separately as in the first two approaches, Cascade and Interleave, makes maintaining and extending the constraints easier. This is an advantage of these approaches. Many natural language processing systems have used these two approaches. For example, Mine proposed a method to represent the morphological constraints in regular grammar and the syntactic constraints in CFG, and interleave the morphological and syntactic analysis (Mine et al., 1990). Most other systems use a connection matrix instead of a regular grammar (Miyazaki et al., 1984;

Sugimura et al., 1989). The main drawbacks of these approaches are as follows:

- It may require two different algorithms for each analysis.
- It must retain all ambiguities from the morphological analysis until the syntactic analysis begins. This wastes memory space and computing time.

On the other hand, from a viewpoint of processing, it is preferable to integrate the morphological and syntactic analysis into a single framework, since some syntactic constraints are useful for morphological analysis and vice versa. The last approach fulfills this requirement. There have been several attempts to develop CFG that covers both the morphological and syntactic constraints (Kita, 1992; Sano–Fukumoto, 1992). However, it is empirically difficult to describe both constraints by using only CFG. The difficulty arises due to the timing of connectability checks, but also increases the number of CFG rules. For example, in figure 1, in order to check the connectability between adjacent words, w_i and w_{i+1}, the morphological attributes of each word should be propagated up to their mother nodes B and C, and the check is delayed until the application of the rule A → B C. Therefore, problems such as the possibility of delays in connectability checking and propagation of morphological attributes to upper nodes make the algorithm of connectability checking more complex and can cause difficulties in representing morphological and syntactical constraints by CFG.

However, by using connection matrices for morphological analysis as in the Cascade/Interleave approaches, connectability checks between adjacent words is performed very easily. Therefore, it is desirable to represent the morphological and syntactic constraints separately as in Cascade/Interleave, and to integrate the execution of both analysis into a single process as in Single Framework. In our method, we have captured these advantages by representing the morphological constraints in connection matrices and the syntactic constraints in CFGs, then compiling both constraints into an LR table (Aho et al., 1986). The already existing, efficient LR parsing algorithms can be used with minor modifications, enabling us to utilize both the morphological and syntactic constraints at the same time.

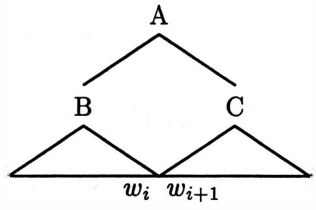


Fig. 1 Connectability check by CFG

In the next section, we first give a brief introduction to Japanese morphological analysis using an example sentence. In section 3, we describe the method of generating an LR table from a connection matrix and CFG rules, then in section 4 we explain the detail of our method based on generalized LR parsing algorithm with an example. Our algorithm is principally the same as Tomita's generalized LR parsing algorithm (Tomita, 1986), but the input is not a sequence of preterminals, but a sequence of characters.

2 Morphological analysis of Japanese

A simple Japanese sentence consists of a sequence of postpositional phrases (PPs) followed by a predicate. The PP consists of a noun phrase (NP) followed by a postposition which indicates the case role of the NP. The predicate consists of a verb or an adjective, optionally followed by a sequence of auxiliary verbs (Morioka, 1987).

We illustrate the Japanese morphological analysis with an example sentence "KaORuNiAIMaSu (meet Kaoru)." [1] We use a simple Japanese dictionary shown in figure 2, and a connection matrix shown in figure 3 which gives us the connectabilities between adjacent morphological categories (mcat). For example in figure 3, the symbol "o" at the intersection of row 2 (`p1`) and column 3 (`vs4k`) indicates that the morphological category `vs4k` can immediately follow the morphological category `p1`.

[1] Each capitalized one or two character(s) corresponds to a single Japanese character (Kana character).

entry	cat	mcat	meaning
KaO	n	n1	face
KaO	vs	vs4r	smell sweet
Ru	ve	ve4r3	(connect to nominal)
KaORu	n	n1	person's name
Ni	p	p1	(dative)
A	vs	vs4k	open
A	vs	vs4w	meet
Ki	ve	ve4k2	(connect to verb)
I	ve	ve4k2i	(connect to verb)
I	ve	ve4w2	(connect to verb)
Tu	ve	ve4w2t	(connect to verb)
MaSu	ax	ax1	(polite form)
Ta	ax	ax2	(past form)

n: noun, p: case marker, vs: verb stem,
ve: verb ending, ax: aux verb

Fig. 2 A simple Japanese dictionary

		RIGHT												
		n1	p1	vs4k	vs4r	vs4w	ve4k2	ve4k2i	ve4r3	ve4w2	ve4w2t	ax1	ax2	$
	n1		o											
	p1	o		o	o	o								
L	vs4k						o	o						
	vs4r								o					
E	vs4w									o	o			
	ve4k2											o		
F	ve4k2i												o	
	ve4r3													o
T	ve4w2											o		
	ve4w2t												o	
	ax1													o
	ax2													o

Fig. 3 An example of connection matrix

Using only the dictionary, we can obtain the following twelve candidates of segmentation for the sentence "KaORuNiAIMaSu."

	KaO	Ru	Ni	A	I	MaSu
(1)	n1	ve4r3	p1	vs4k	ve4k2i	ax1
(2)	n1	ve4r3	p1	vs4k	ve4w2	ax1
(3)	n1	ve4r3	p1	vs4w	ve4k2i	ax1
(4)	n1	ve4r3	p1	vs4w	ve4w2	ax1
(5)	vs4r	ve4r3	p1	vs4k	ve4k2i	ax1
(6)	vs4r	ve4r3	p1	vs4k	ve4w2	ax1
(7)	vs4r	ve4r3	p1	vs4w	ve4k2i	ax1
(8)	vs4r	ve4r3	p1	vs4w	ve4w2	ax1

	KaORu	Ni	A	I	MaSu
(9)	n1	p1	vs4k	ve4k2i	ax1
(10)	n1	p1	vs4k	ve4w2	ax1
(11)	n1	p1	vs4w	ve4k2i	ax1
(12)	n1	p1	vs4w	ve4w2	ax1

By also referring to the connection matrix, we can filter out illegal segmentations. From the examples above, we find (1)–(4) violate the connectability between "KaO (n1)" and "Ru (ve4r3)", and that (5)–(8) violate the connectability between "Ru (ve4r3)" and "Ni (p1)." Also (9) and (11) violate the connectability between "I (ve4k2i)" and "MaSu (ax1)", and (11) violates the connectability between "A (vs4w)" and "I (ve4k2i)." Thus by process of elimination we obtain the morphologically correct candidate, (12). However, a long input sentence generally gives many more ambiguities which need to be resolved in later stages using syntactic, semantic and contextual information.

3 Generating A Modified LR Table

Connection matrices and CFG rules have been used for morphological analysis and syntactic analysis respectively by most Japanese processing systems. Because CFG rules were mainly used for syntactic analysis and connection matrices for morphological analysis, they have been developed independently of each other.

In this section, we propose a method to integrate morphological and syntactic constraints in the framework of LR parsing algorithm, and thus capturing the advantages of Cascade/Interleave and Single Framework described in section 1.

In order to combine connection matrices and CFG rules, the first step we have to take is to extend the CFG rules by relating the syntactic categories in the CFG rules with the morphological categories in a connection matrix. This is realized by adding CFG rules called morphological rules each of which is a unit production rule with a syntactic category in the LHS and a morphological category in the RHS.

From the dictionary shown in figure 2, we can extract a set of new CFG rules as shown in figure 6, which are simply added to the CFG rules in figure 4 to get an extended set of CFG rules with morphological constraints.

```
s → v ax   (1)    v → vs ve   (3)
s → pp s   (2)    pp → n p    (4)
```

Fig. 4 A simple set of CFG rules for Japanese

	ACTION											GOTO							
state	n 1	p 1	vs 4 k	vs 4 r	vs 4 w	ve 4 k 2	ve 4 k 2 i	ve 4 r 3	ve 4 w 2	ve 4 w 2 t	ax 1	ax 2	$	s	v	ax	pp	vs	ve	n	p
0	sh6		sh7	sh8	sh9									1	2		3	4		5	
1													acc								
2											sh11	sh12				10					
3	sh6		sh7	sh8	sh9									13	2		3	4		5	
4						sh15	sh16	sh17	sh18	sh19									14		
5		sh21																			20
6		re5																			
7						re7	re7	re7*	re7*	re7*											
8						re8*	re8*	re8	re8*	re8*											
9						re9*	re9*	re9*	re9	re9											
10													re1								
11													re15								
12													re16								
13													re2								
14											re3	re3									
15											re10	re10*									
16											re11*	re11									
17											re12*	re12*									
18											re13	re13*									
19											re14*	re14*									
20	re4		re4	re4	re4																
21	re6		re6	re6	re6																

Fig. 5 LR table generated from rules (1)–(16)

n → n1	(5)	ve → ve4k2i	(11)
p → p1	(6)	ve → ve4r3	(12)
vs → vs4k	(7)	ve → ve4w2	(13)
vs → vs4r	(8)	ve → ve4w2t	(14)
vs → vs4w	(9)	ax → ax1	(15)
ve → ve4k2	(10)	ax → ax2	(16)

Fig. 6 A morphological rules derived from the dictionary in Fig. 2

We can generate an LR table as shown in figure 5 from the extended CFG rules (1) through (16) from figure 4 and 6. Note that the extended CFG rules do not include any information about connectability represented in the connection matrix in figure 3. For example, rules (3), (8) and (13) allow the structure "v(vs(vs4r),ve(ve4w2))" which violates the connectability between vs4r and ve4w2 as shown in figure 3.

For each reduce action **A** with a morphological rule in each entry of LR table {
  if (Not *Connect*(*RHS*(*Rule*(**A**)), *LA*(**A**)) {
    delete **A** from the entry;
  }
}

where each function is defined as follows:

Rule : *action* → *rule*;
  returns a rule used by the reduce action.

LA : *action* → *symbol*;
  returns a look ahead symbols of the action.

Connect : *symbol* × *symbol* → {*T*, *F*};
  returns true or false with respect to the connectability of the two symbols.

RHS : *rule* → *symbol*;
  returns a right hand side symbol of the rule.

Fig. 7 A procedure to modify an LR table

The second step is to introduce the constraints on connectability into the LR table by deleting illegal reduce actions. This is carried out by modifying the LR table with the procedure shown in figure 7.

Deleting reduce actions by applying the above procedure prohibits the application of morphological rules which violates the connectability between two adjacent words, namely the current scanned word and its lookahead word. Note that

given an LR table and a connection matrix, this procedure can be performed automatically without human intervention.

It is possible to incorporate this procedure into the LR table generation process, however, it is better to keep them separate. Since this procedure is applicable to any type of LR table, separating this process from LR table generation enables us to use the already existing LR table generation program.

For example, in figure 5, the reduce action `re7` in row 7 and column `ve4r3` is deleted, since the connection between `vs4k`, the RHS of rule (7), and `ve4r3`, the lookahead preterminal, is prohibited as shown in the connection matrix in figure 3. Similarly, reduce action `re7` in row 7 and column `ve4w2` will be deleted and so forth. These deletions are marked with asterisks (*) in figure 5. The overview of generating a modified LR table is shown in figure 8.

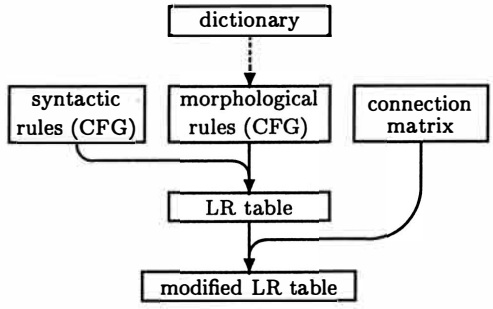


Fig. 8 Outline of generating a modified LR table

Generally speaking, the size of the LR table is on the exponential order of the number of rules in the grammar. Introducing the morphological rules into the syntactic rules can cause an increase in the number of states in LR table, thereby exponentially increasing the size of the overall LR table in the worst case. In our method, the increase of the number of states is equal to that of the morphological rules introduced. Suppose we add a morphological rule $X \rightarrow x$ to the grammar. Only the items in the form of $[A \rightarrow \alpha \cdot X\beta]$ can produce a single new item $[X \rightarrow \cdot x]$ from which only a single new state $\{[X \rightarrow x\cdot]\}$ can be created. Thus the increase of the number of the states is equal to that of the morphological rules introduced, and the size of the LR table will not grow exponentially.

```
(1)  initialize stack
(2)  for CS = 0 ... N {
(3)    for each stack top node in stage CS {
(4)      Look-aheads = lookup-dictionary(CS);
(5)      for each look ahead preterminal LA
           in Look-aheads {
(6)        do reduce while "reduce" is applicable;
(7)        if "shift" is applicable {
(8)          do shift creating a new node
               in stage (CS + length(LA));
(9)        }
(10)       if "acc" { accept }
(11)       if no action { reject }
(12)     }
(13)   }
(14) }
```

Fig. 9 Outline of our parsing algorithm

4 Algorithm for Integrating Morphological and Syntactic Analysis

The LR parsing algorithm with the modified LR table is principally the same as Tomita's generalized LR parsing algorithm. The only difference is that Tomita's algorithm assumes a sequence of preterminals as an input, while our algorithm assumes a sequence of Kana characters[2]. Thus the dictionary reference process needs to be slightly modified. Figure 9 illustrates the outline of our parsing algorithm.

In figure 9 the stage number (CS) indicates how many Kana characters have been processed. The procedure begins at stage 0 and ends at stage N, the length of an input sentence. In stage 0, the stack is initialized and only the node with state 0 exists (step (1)). In the outer-most loop (2)–(14), each stack top in the current stage is selected and processed. In step (4), the dictionary is consulted and look-ahead preterminals are obtained. An important point here is that look-ahead preterminals may have different Kana character lengths. A new node is introduced by a shift action at step (8) and is placed into a stage which is ahead of the current stage by the length of the look-ahead word.

The following example well illustrates the algorithm in figure 9. The input sentence is

[2]We assume the input sentences consist of only Kana characters for brevity. Other types of characters, such as Kanji, can be also handled.

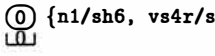
"KaORuNiAIMaSu$ (meet Kaoru)." and we assign position numbers between adjacent Kana characters.

Input:	Ka	O	Ru	Ni	A	I	Ma	Su	$
Position:	0	1	2	3	4	5	6	7	8 9

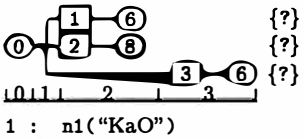
In the following trace, the numbers in circles denote state numbers, and the numbers in squares denote the subtree number shown below the diagrams. The symbols enclosed by curly brackets denote a look ahead preterminal followed by the next applicable action, separated by a slash (/). The stage numbers are shown below the stacks.

Current stage: 0
Dictionary reference:
n1("KaO") at 0–2
vs4r("KaO") at 0–2
n1("KaORu") at 0–3

We find three look ahead preterminals, **n1**, **vs4r**, and **n1** by consulting the dictionary in figure 2. A shift actions is applied for each of them according to the LR table in figure 5.



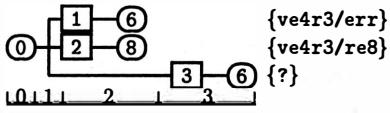
After the shift actions, three new nodes are created at stage 2 or stage 3 depending on the length of look ahead words. At the same time subtrees 1–3 are constructed. The current stage is updated from 0 to 2, since there is no node in stage 1. The look ahead preterminals are unknown at this moment.



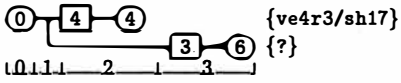

```
1 :  n1("KaO")
2 :  vs4r("KaO")
3 :  n1("KaORu")
```

Current stage: 2
Dictionary reference:
ve4r3("Ru") at 2–3

Dictionary reference gives one look ahead preterminal from position 2. Since the current stage number is 2, only the first two stack tops are concerned at this stage. No action is taken of the first stack, because the LR table has no action in the entry for state 6 and a look ahead preterminal **ve4r3**. As the result, the first stack is rejected. The reduce action (**re8**) is taken for the second stack.

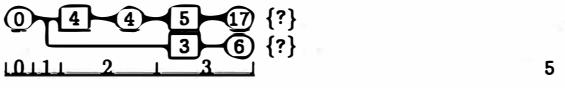


After **re8**, a shift action (**sh17**) is carried out for the first stack.




```
4 :  vs(2)
```

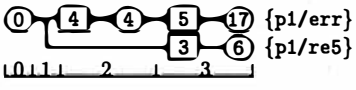
After **sh17**, we can proceed to stage 3.



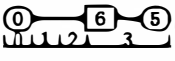

```
:  ve4r3("Ru")
```

Current stage: 3
Dictionary reference:
p1("Ni") at 3–4

We obtain preterminal **p1** by consulting the dictionary. Because the first stack can take no more action, it is rejected. The reduce action (**re5**) is then applied to the second stack.

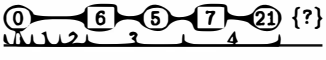


The shift action (**sh21**) is applied to the following stack.




```
6 :  n(3)
```

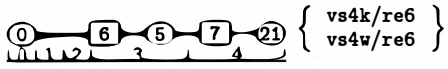
After the shift action (**sh21**), new nodes are created in stage 4.



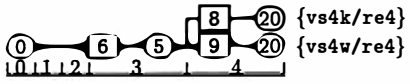

```
7 :  p1("Ni")
```

Current stage: 4
Dictionary Reference:
vs4k("A") at 4–5
vs4w("A") at 4–5

Dictionary reference provides two look ahead preterminals for the next word.

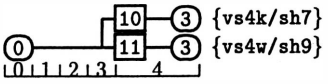


After the two reduce actions (re6), we get two nodes with the same state 20, but they are not merged as the look ahead preterminals are different each other. See stage 5 for the reason.

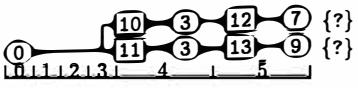



```
8 :  p(7)
9 :  p(7)
```

The process in stage 4 continues as follows.



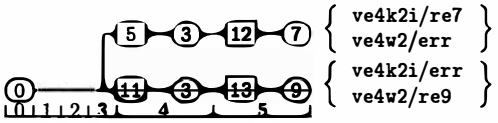

```
10 :  pp(6,8)
11 :  pp(6,9)
```



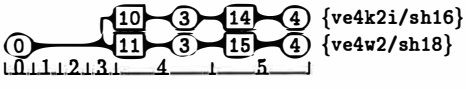

```
12 : vs4k("A")
13 : vs4w("A")
```

Current stage: 5
Dictionary Reference:
ve4k2i("I") at 5–6
ve4w2("I") at 5–6

We have two look ahead preterminals and two stack tops. The reduce actions (re7 and re9) are performed.

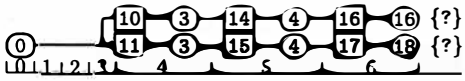


Note that we can not merge the stack tops with the same state 4 since the look ahead preterminals are different (ve4k2i/ve4w2). [3]




```
14 :  vs(12)
15 :  vs(13)
```

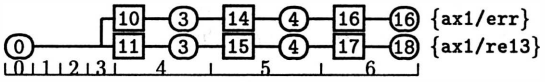
After the shift actions (sh16 and sh18), we proceed to stage 6.

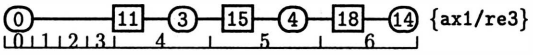



```
16 :  ve4k2i("I")
17 :  ve4w2("I")
```

Current stage: 6
Dictionary reference:
ax1("MaSu") at 6–8

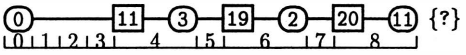
The process in stage 6 proceeds as follows.





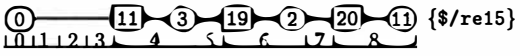

```
18 :  ve(17)
```


```
19 :  v(15,18)
```

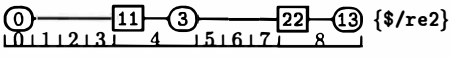



```
20 :  ax1("MaSu")
```

Current stage: 8
Dictionary reference:
"$" at 8–9

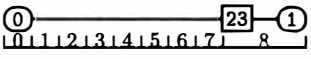



```
21 :  ax(20)
```




```
22 :  s(19,21)
```

The input sentence is automatically segmented and accepted, giving a final parse result 23 as shown in figure 10.




```
23 :  s(11,22)
```

[3] If two stack tops are merged and then different shift actions (sh16 and sh18) are carried out, we might have invalid combinations of structure such as (14, 17) and (15, 16).

5 Conclusion

We have proposed a method representing the morphological constraints in connection matrices and the syntactic constraints in CFGs, then compiling both constraints into an LR table. The compiled LR table enables us to make use of the already existing, efficient generalized LR parsing algorithms through which integration of both morphological and syntactic analysis is obtained.

Advantages of our approach can be summarized as follows:

- Morphological and syntactic constraints are represented separately, and it makes easier to maintain and extend them.
- The morphological and syntactic constraints are compiled into a uniform representation, an LR table. We can use the already existing efficient algorithms for generalized LR parsing for the analysis.
- Both the morphological and syntactic constraints can be used at the same time during the analysis.

We have implemented our method using the EDR dictionary with 300,000 words (EDR, 1993) from which 437 morphological rules are derived. This means only 437 new states are introduced to LR table and this does not cause an explosion in the size of the LR table.

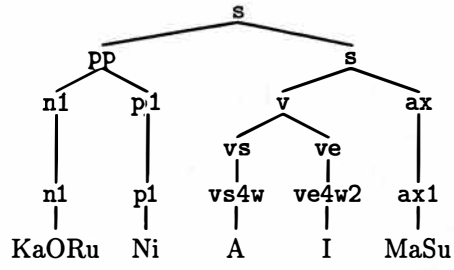


Fig. 10 An analysis of "KaORuNiAIMaSu"

References

Aho, A.V. — Sethi,R. — Ullman, J.D. (1986) *Compilers Principles, Techniques, and Tools.* Massachusetts: Addison-Wesley.

Japan Electronic Dictionary Research Institute (1993) *EDR Dictionary Manual.*

Kita, K. (1992) *A Study on Language Modeling for Speech Recognition.* PhD thesis, Waseda University.

Mine,T. — Taniguchi, R. — Amamiya, M. (1990) "A parallel syntactic analysis of context free grammars." pp. 452–453. the 40th Annual Convention IPS Japan.

Miyazaki, M. (1984) "An Automatic Segmentation Method for Compound Words using Dependency Analysis." Transactions of Information Processing Society of Japan. Vol. 25, No. 6, pp. 970–979

Morioka, K. (1987) *Formation of a Vocabulary.* Meiji-Shoin.

Sano, H. — Fukumoto, F. (1992) "On a Grammar Formalism, Knowledge Bases and Tools for Natural Language Processing in Logic Programming." in Proceedings of FGCS92.

Sugimura,R. — Akasaka, K. — Kubo, Y. — Matsumoto, Y. (1989) "Logic Based Lexical Analyzer LAX." Logic Programming '88 (Lecture Notes in Artificial Intelligence). pp. 188–216. Springer-Verlag.

Tomita, M. (1986) *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Boston: Kluwer Academic Publishers.

Structural Disambiguation in Japanese by Evaluating Case Structures based on Examples in a Case Frame Dictionary

Sadao Kurohashi and Makoto Nagao

Dept. of Electrical Engineering, Kyoto University
Yoshida-honmachi, Sakyo, Kyoto, 606, Japan
email: {kuro|nagao}@kuee.kyoto-u.ac.jp

Abstract

A case structure expression is one of the most important forms to represent the *meaning* of a sentence. Case structure analysis is usually performed by consulting *case frame information* in verb dictionaries and by selecting *a proper case frame* for an input sentence. However, this analysis is very difficult because of *word sense ambiguity* and *structural ambiguity*. A conventional method for solving these problems is to use the method of *selectional restriction*, but this method has a drawback in the semantic marker (SM) system — the trade-off between descriptive power and construction cost.

This paper describes a method of case structure analysis of Japanese sentences which overcomes the drawback in the SM system, concentrating on the structural disambiguation. This method selects a proper case frame for an input by the similarity measure between the input and typical example sentences of each case frame. When there are two or more possible readings for an input because of structural ambiguity, the best reading will be selected by evaluating case structures in each possible reading by the similarity measure with typical example sentences of case frames.

1 Introduction

Representing a sentence with a case structure is a basic form for dealing with its *meaning*. Therefore, transforming a sentence into a case structure expression is one of the most important techniques in natural language processing, and it is needed for machine translation, knowledge acquisition in which various expressions with the same content must be converted into the same representation, and so on.

Case structure analysis is usually performed by consulting *case frame information* in verb dictionaries. The dictionary describes what kind of cases each verb has and what kinds of noun can fill a case slot with what kind of case marker (in Japanese, postpositions (POs) function as case markers). However, this analysis is very difficult because of *word sense ambiguity* (a verb often has two or more meanings and case frames are prepared for the respective meanings) and *structural ambiguity*.

A conventional method for solving these problems is *selectional restriction* (Katz — Fodor, 1963), where the category of the nouns which are able to fill in the case slot is specified by semantic markers (SMs), such as *human*, *animate*, *action*, and so on. However this method has the following weak points.

- The SM system with tens of SMs is too coarse to distinguish every case frame for a verb, which is the case with most electronic dictionaries and systems at present, such as LDOCE (Longman, 1978), the Mu system (Nagao et al., 1985), IPAL (IPA, 1987), and most of commercial machine translation systems.

HAIRU			
Sub-entry 1			
Meaning: Enter from the outside to the inside.			
<Case markers>	*<SMs>*	*<Examples>*	*<Deep cases>*
N1-GA	[HUM/ORG/ANI/PRO]	he party cat ship	agent
N2-KARA*	[LOC]	window rear-gate	locational source
N3-NI/E	[LOC]	classroom kitchen port	locational goal / directional
Sub-entry 2			
Meaning: Added to food or drink.			
<Case markers>	*<SMs>*	*<Examples>*	*<Deep cases>*
N1-NI	[PRO]	coffee cake	non-locational goal
N2-GA	[CON]	sugar milk cheese poison	object
Sub-entry 3			
Meaning: Be reflected.			
<Case markers>	*<SMs>*	*<Examples>*	*<Deep cases>*
N1-NI	[PRO/ABS]	work report proposal	non-locational locative
N2-GA	[MEN]	thought opinion arbitrariness	object

A Case component marked with '*' is optional.

Table 1: Examples of case frames for HAIRU in IPAL.

- On the other hand, it is quite expensive and time consuming to prepare a detailed SM system which has enough descriptive power to discriminate every usage of each verb, and which may require thousands of SMs (Ikehara et al., 1991). A further difficulty is to improve the SM system when needed.

In order to overcome the drawbacks in the SM method — the trade-off between descriptive power and construction cost, we have developed a method of case structure analysis of Japanese sentences based on examples in a case frame dictionary. We published some parts of our analysis system already elsewhere (Kurohashi — Nagao, 1992) (Nagao, 1992). Therefore, this paper concentrates on the structural disambiguation in Japanese complex sentences through the case structure analysis process.

This method uses a case frame dictionary that has some typical example sentences for every case frame. When an input is a simple sentence without structural ambiguity concerning case components, a proper case frame is selected for the verb in the input sentence by matching the input sentence with the examples in the case frame dictionary. When an input is a complex sentence, there would be several verbs to which the nouns in the sentence cannot be linked uniquely by unique case assignment. The important point is that the best matching score, which is utilized for selecting a proper case frame for a verb in a sentence, can be considered as *the score for the case structure* of the verb and its case components. The best reading (the correct reading or the most plausible reading) of a complex sentence is the one where all verbs in the sentence govern appropriate case components and their case structures have high scores. Therefore, the best reading of a sentence can be selected by checking all the possible case structures of all the verbs and by evaluating every possible reading according to the sum of the scores for the case structures in it. When an input is a compound sentence, we can detect the scopes of coordinate structures beforehand, so that we can limit the possible readings to the extent that we can evaluate all of them.

From the viewpoint of an example-based method, there are several research activities for solving structural ambiguity (Inagaki et al., 1988) (Nagao, 1990). Our method has the following characteristics in contrast with them.

- While their methods match an input sentence with examples basically in blocks of two words being in a governor/dependent relation, our method matches them in

blocks of a case structure. We can say in general that the wider range of components a method checks, the more reliable it becomes.

- They use texts from their target domain as their main knowledge base. If limiting the text domain, it may be possible and useful, but it is very difficult to cover general domains. On the other hand, we use examples in the case frame dictionary, which can cover wider domains according to entries of the dictionary. Because in compiling a dictionary, lexicographers consult example sentences in which an entry word is used, it is a reasonable assumption that we can use examples in a dictionary in the computer analysis of natural language sentences.

2 Selecting a Proper Case Frame

2.1 Japanese Electronic Dictionaries

In this paper, we use the basic verb dictionary which was constructed by the Information-technology Promotion Agency, Japan (hereafter, this dictionary is referred to as IPAL) (IPA, 1987). In IPAL, 861 basic verbs are entries, and each entry has sub-entries according to the difference in the meaning and the syntax. Case frame information is given at each sub-entry. The sub-entries total up to 3379 so that the average number of sub-entries and thus the average number of case frames for a verb is 3.9. As shown in Table 1, a set of case frame information consists of the meaning of a verb, its case markers, SMs, examples and correspondences to deep cases for each case slot. SMs restrict the category of the nouns that can fill in the case slots. IPAL uses 19 different SMs which have the tree structure shown in Figure 1. However, our method does not use this SM system because of its coarseness. We use a thesaurus dictionary, 'Bunrui Goi Hyou' (abbreviated as BGH) (NLRI, 1964) for calculating the similarity values between words. BGH has a tree of six layer abstraction hierarchy and more than 60,000 words are assigned to the leaves of the thesaurus tree.

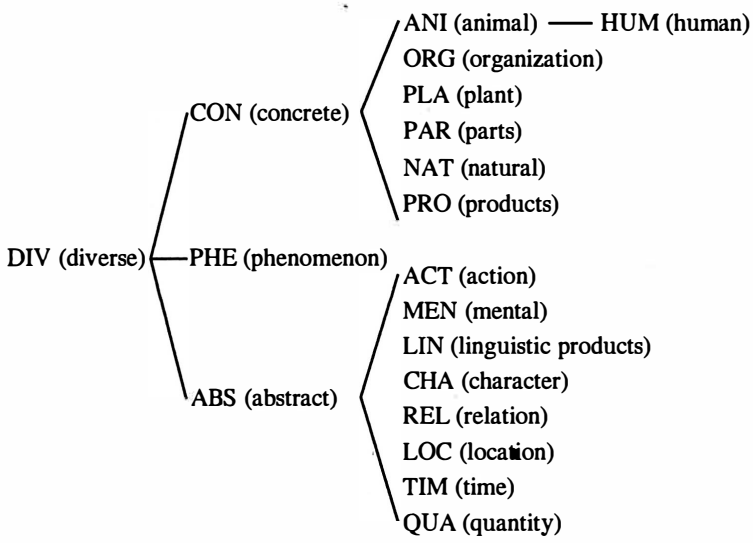


Figure 1: The set of SMs in IPAL.

2.2 Case Structure Analysis by Case Frames

Case structure analysis is usually performed by consulting case frames in a dictionary. Because the correspondence of each case component to a deep case is listed in the case frame, we can get deep cases for case components when we find a proper case frame for an input sentence and the correspondence of case components in the sentence to those in the case frame. In other words, case structure analysis is regarded as a selection of a proper case frame for an input sentence.

2.3 Selecting a Proper Case Frame

A conventional method for selecting a proper case frame is selectional restriction by SMs. However, the SM system with tens of SMs, such as IPAL, is too coarse to select a proper case frame for an input sentence. For instance, in selecting a proper case frame for the example sentence (ES1):

KISHA-GA	TON'NERU-NI	HAIRU.
(train)	(tunnel)	(enter)
[PRO]	[LOC/PRO]	

out of the case frames in Table 1, while sub-entry 3 can be removed by comparing the SM [PRO(product)] of 'KISHA(train)' with the SM [MEN(mental)] of case slot 'N2-GA', the incorrect case frame, sub-entry 2, is selected together with the correct case frame, sub-entry 1. On the other hand, it is quite expensive and time consuming to prepare a detailed SM system which has enough descriptive power to discriminate every usage of each verb.

In order to overcome this drawback in the SM method, we have developed a method for case structure analysis of Japanese sentences based on examples in a case frame dictionary. In brief, this method selects the case frame whose example is the most similar to the input sentence. Because ES1 above is much more similar to examples of sub-entry 1, "(he party cat ship)-GA (classroom kitchen port)-NI", than to examples of sub-entry 2, "(coffee cake)-NI (sugar milk cheese poison)-GA", the correct case frame, sub-entry 1, can be selected.

The similarity score between an input sentence and examples of a case frame is calculated by the following algorithm. The algorithm assumes that case components depending on a verb are already known, as in the case of processing a simple sentence.

1. **Matching case components.**
   First, case components of the input sentence and those of a case frame are matched by the equality of POs. A noun modified by a clause sometimes becomes a case component for the verb of the modifying clause. In this case, the modified noun can correspond to case slots followed by PO 'GA', 'WO', 'NI' or 'DE'.

2. **Calculating the score of a matching case component.**
   A score of matching case components is defined as the greatest similarity value between a noun of the input sentence and example nouns assigned to a case slot in the case frame dictionary. The similarity value (SV) between two nouns is given according to the most specific common layer (CL) between them in BGH, as follows:[1]

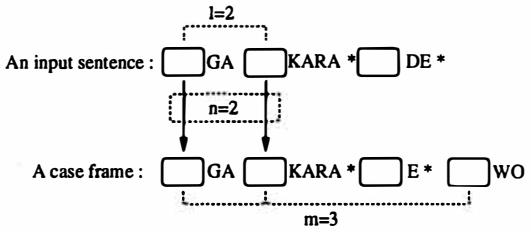
CL	0	1	2	3	4	5	6	exact match
SV	0	0	5	7	8	9	10	11

3. **Calculating the score of a matching pattern.**
   It is assumed that *the matching pattern* between an input sentence and a case frame is given as follows:

[1] This way of correlating the most specific common layer (CL) with the similarity value has the following basis ($sv(i)$ means the similarity value between two nouns whose CL is i).

(a) Since the first layer of BGH consists of four classes: nouns, verbs, adjectives and the others, sharing the first layer of two nouns does not indicate that they are similar. Therefore, we let $sv(1)$ '0'.

(b) The greater the CL between two nouns is, the more similar they are. Furthermore, by studying BGH, we concluded that sharing the general layer (except the first layer) has more effect on the similarity between two nouns than sharing the specific layer. For this reason, $sv(i)$ is designed to simulate a convex and monotone increasing function.



n : the number of matching components.

l : the number of matching or obligatory case components in the input sentence. Case components followed by PO 'GA', 'WO', 'NI', 'E' or 'YORI', are regarded as obligatory case components.

m : the number of matching or obligatory case slots in the case frame. Obligatory case slots are specified in IPAL.

total_score : the sum of scores of matching case components.

The simplest way is to regard the *total_score* as the score of this matching pattern. However, we need to take more factors into consideration. We give the following score to this matching pattern:

$$\begin{cases} \text{if } l > n & 0 \\ \text{otherwise} & \textit{total\_score} \times \left(\frac{1}{n}\right)^{1/2} \times \left(\frac{n}{m}\right)^{1/2} \end{cases}$$

We let the score '0' when l is greater than n, because the obligatory case components cannot remain unmatched in the sentence for its proper case frame. We include $(n/m)^{1/2}$ in the above formula in order to give priority to case frames with the higher ratio of matching case components to the total number of case components. We also include $(1/n)^{1/2}$ because it is preferable not only that there are many matching components but also that the scores of matching components are big. The exponents of $(n/m)^{1/2}$ and $(1/n)^{1/2}$ were determined empirically.

Matching calculation is performed for all the matching patterns between the input sentence and all case frames, and then the case frame whose matching pattern has the greatest score is selected as the final result.

The experiment of comparing the example based method with the SM method in IPAL, and the discussion about the validity of the example based method were reported in (Nagao, 1992).

3 Structural Disambiguation using Case Structure Score

3.1 Outline of the Method

In the preceding section we described a method of analyzing the case structure for an input sentence when case components depending on a verb are already known, as in the case of a simple sentence. This section introduces the way of extending the method to process complex or compound sentences.

Japanese sentences can best be explained by "Kakari-uke", which is essentially the governor/dependent relation between *bunsetsus*.[2] A bunsetsu depends on, that is, modifies another bunsetsu to its right (not necessarily the adjacent bunsetsu). Sometimes a bunsetsu can depend on two or more bunsetsus, which creates structural ambiguity and makes case structure analysis hard. Other work concerning structural disambiguation (Inagaki et al., 1988) (Nagao, 1990) solves this problem locally, that is, they try to determine the governor of each bunsetsu independently. However, in order to improve the precision of analyzing sentences, the ambiguity of the governor of a bunsetsu must be processed simultaneously with the ambiguity of the governors of other bunsetsus and with the word sense ambiguity.

[2]Bunsetsu is the smallest meaningful block consisting of an independent word (nouns, verbs, adjectives, etc.) and accompanying words (POs, auxiliary verbs, etc.).

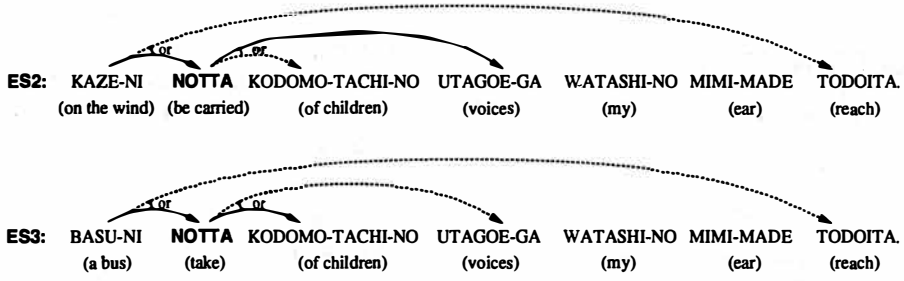


Figure 2: Need for a global disambiguation.

The ambiguity in example sentences in Figure 2 makes this problem clear. In ES2, the meaning of the verb 'NOTTA' is 'be carried (on the wind)', and 'KAZE-NI(on the wind)' depends on 'NOTTA(be carried)' and 'NOTTA(be carried)' depends on 'UTAGOE-GA(voices)'; whereas, in ES3, the meaning of the same verb 'NOTTA' is 'take (a bus)', and 'BASU-NI(a bus)' depends on 'NOTTA(take)' and 'NOTTA(take)' depends on 'KODOMO-TACHI-NO(of children)'. This means that whether 'NOTTA' depends on 'KODOMO-TACHI-NO(of children)' or 'UTAGOE-GA(voices)' can not be determined independently of the structural ambiguity of other case components of the verb 'NOTTA' and its word sense ambiguity.

For such a global disambiguation, we can use the best matching score which is utilized for selecting a proper case frame for each verb. The best matching score between an input sentence and a typical example for the usage of a verb can be considered as *the appropriateness* (score) for the case structure of the verb and its case components. When there are two or more readings (dependency structures) for a sentence because of structural ambiguity, the best reading (the correct reading or the most plausible reading) is the one where all verbs in the sentence govern appropriate case components and their case structures have high scores. This means that the best reading of a sentence can be selected by evaluating the sum of the scores for the case structures of all verbs in a sentence (Figure 3).

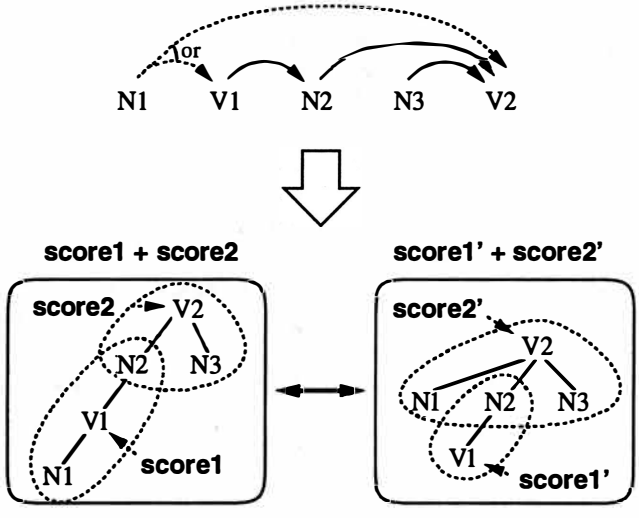


Figure 3: Outline of the method.

1	**Key bunsetsu of coordinate structure (KB)**	Bunsetsu which indicates the existence of a coordinate structure, such as "...SHI", and "...TO". This type is treated as depending on the last bunsetsu in the coordinate structure.
2	**PB depending on PB**	PB in a kind of subordinate structure, such as "...SURE-BA (if...V)", "...SITA-NODE (because ...V)".
3	**NB depending on PB**	Case components, such as "...GA", "...WO".
4	**PB depending on NB**	PB in a clausal modifier, such as "...SITA (which ...=V)". A governor of this type of a bunsetsu may become a case component for the bunsetsu.
5	**NB depending on NB**	NB, such as "A-NO" in the noun phrase "A-NO B".

Table 2: Dependent types.

In order to evaluate all the possible readings, it is necessary to expand all the structural ambiguity for a sentence. However, before entering this analysis stage, we detect the scope of coordinate structures in the sentence by using another method (Kurohashi — Nagao, 1992) to avoid the combinatorial explosion problem. The main reason that a sentence becomes long, particularly in Japanese, is that two or more matters are expressed in a sentence, that is, a sentence has coordinate structures. Therefore, by detecting the scopes of coordinate structures beforehand, the possible readings are limited to the extent that we can evaluate all of them.

We will explain this method in detail in the following subsections.

3.2 Calculation of the Possible Dependency Matrix (PDM)

For evaluating each possible dependency structure, we first get all the possible governor/dependent relations between two bunsetsus. These relations are expressed in the form of a triangular matrix $A = (a_{ij})$ (Figure 4), called possible dependency matrix (PDM), whose diagonal element a_{ii} is the i-th bunsetsu (hereafter expressed as B_i) in a sentence and whose element a_{ij} (i < j) expresses whether B_i can depend on B_j ('1' means yes).

As a governor, each bunsetsu is classified into one of two types according to parts of speech; nominal bunsetsu (abbreviated as NB) and predicative bunsetsu (abbreviated as PB). As a dependent, each bunsetsu is classified into one of five types in Table 2 according to its PO or conjugation.

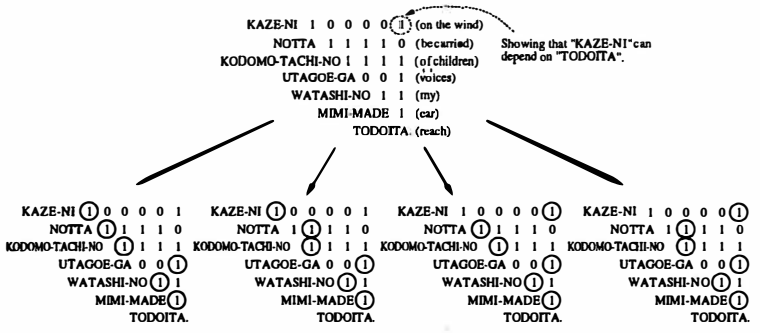


Figure 4: Making the possible dependency structures by consulting the PDM (ES2).

The way of determining an element a_{ij} is rather simple. The value of element a_{ij} is adjusted to '1' when B_i is type 2 or type 3 (which can depend on PB) and B_j is a PB, or when B_i is type 4 or type 5 (which can depend on NB) and B_j is a NB.

3.3 Masking the PDM by the Scope of the Coordinate Structure

When an input sentence contains a key bunsetsu of coordinate structure (abbreviated as KB) which indicates the existence of a coordinate structure, we detect its scope by the method in which the two most similar series of bunsetsus on the left and right side of the KB are detected and are regarded as the scope of the coordinate structure concerning the KB (see (Kurohashi — Nagao, 1992) for details).

After detecting the scopes of coordinate structures in a sentence, the following two operations are performed on the PDM (Figure 5).

- *Setting the governor of a KB*: A KB is treated as depending on the last bunsetsu of a coordinate structure. When B_i is a KB and B_j is the last bunsetsu of its scope, the value of the PDM element a_{ij} is adjusted to '1'.
- *Masking the PDM*: Because the prior and the posterior parts of a coordinate structure have their own consistent structures and meanings, they are parsed independently into dependency structures. Therefore, the bunsetsu in a coordinate structure does not depend on or become governor of any bunsetsu outside its scope, except the last bunsetsu of the coordinate structure which has governor/dependent relations to bunsetsus outside its scope. In order to express these characteristics, the value of the PDM elements on the upper and right side of a coordinate structure are set to '0'.

As a result of this process, the number of possible dependency structures of a sentence can be reduced drastically.

3.4 Making the Possible Dependency Structures

Next, we expand the ambiguities of the case components of an input sentence and make the possible dependency structures by consulting the PDM. Because governor/dependent relations do not cross each other in Japanese, *no-cross condition* can be set as follows: when B_i depends on B_j, B_k (k < i) cannot depend on bunsetsus from B_{i+1} to B_{j-1}. For the following explanation, we

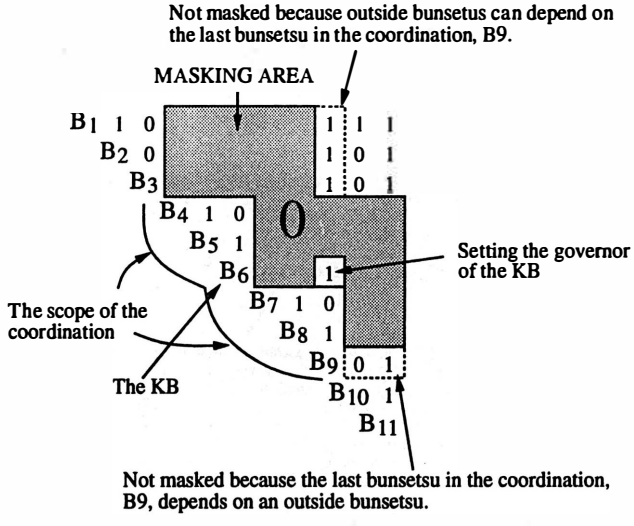


Figure 5: Masking the PDM.

define *a dependency set* as a set of a bunsetsu consisting of bunsetsus on which the bunsetsu can depend. This set is fixed dynamically by the PDM and the no-cross condition.

The governor is determined for each bunsetsu from right to left. When a bunsetsu concerning a case component (type 3 or type 4 in Table 2) has the possibility of depending on two or more bunsetsus (that is, its dependency set consists of two or more bunsetsus), two or more partial dependency structures are made according to the varieties of its governor, and the structures on the left side of each partial structure are analyzed. On the other hand, when a bunsetsu, not concerning case components (type 2 or type 5), can depend on two or more bunsetsus, its governor is determined uniquely to be the nearest bunsetsu in its dependency set, because a bunsetsu usually depends on the nearest bunsetsu in Japanese (of course, this heuristic rule sometimes makes a mistake. We will deal with this problem in future work).

In the case of ES2 (Figure 4 on page 117), because 'NOTTA (be carried)' can depend on either 'KODOMO-TACHI-NO (of children)' or 'UTAGOE-GA (voices)' and 'KAZE-NI (on the wind)' can depend on either 'NOTTA (be carried)' or 'TODOITA (reach)', four possible dependency structures are created.

3.5 Evaluation of Possible Dependency Structures

Case frame selection is performed for all verbs in each possible dependency structure which is made in the above-mentioned processes. Of all possible dependency structures, we select the structure which has the maximum sum of the best matching scores for all verbs in the sentence. In the case of ES2, the correct structure, the second one form the left in Figure 4, is selected by this method, selecting proper case frames for verbs 'NOTTA' and 'TODOITA' (the sub-entry whose meaning is "be carried" is selected for 'NOTTA' correctly).

If there are two or more structures which have the maximum score, the structure which is most similar to the *default dependency structure* is selected. Here, a default dependency structure is that in which each bunsetsu depends on its nearest bunsetsu in its dependency set.

Our method	MT systems	Type3	Type4	Total
○	○ ×	56	12	68
○	× ×	23	15	38
×	○○	5	0	5
×	○×	6	0	6
○	○○	104	6	110
×	× ×	2	4	6

Table 3: Comparison between our method and commercial MT systems.

4 Experiment

We report a experiment which illustrates the effectiveness of this method for solving structural ambiguity. This method limits the possible readings by detecting coordinate structures beforehand; the validity of the method for detecting coordinate structures has already been reported in (Kurohashi — Nagao, 1992). After detecting coordinate structures, the remaining problem is the ambiguity in a complex sentence. Therefore, in this paper, we show an experiment of analyzing complex sentences.

We had a language-trained person compose a set of about 450 complex test sentences each of which includes one or more clausal modifier. Then we analyzed these test sentences by our method and evaluated the analysis results from the viewpoint of structural disambiguation according to the following structural types of sen-

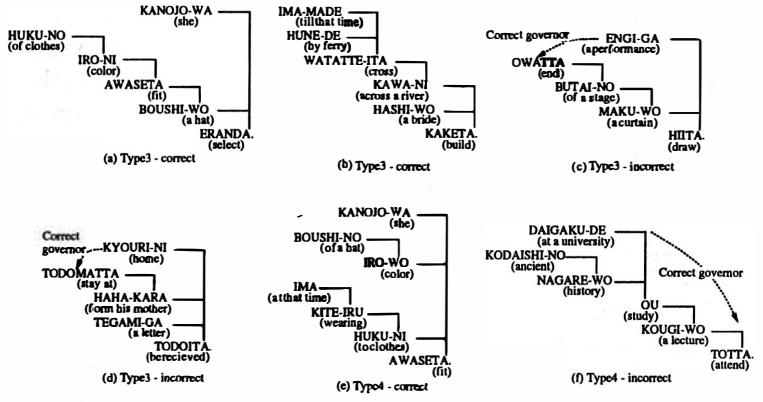


Figure 6: Examples of detecting dependency structures.

tences.

Type 1 : A sentence which has no structural ambiguity concerning case components.

Type 2 : A sentence which has two or more correct dependency structures.

Type 3 : A sentence which has two or more possible dependency structures and whose correct dependency structure is its default dependency structure.

Type 4 : A sentence which has two or more possible dependency structures and whose correct dependency structure is not its default dependency structure.

The left part of Table 4 shows its results. This table shows that the success ratio of getting a correct dependency structure by this method is very high. Examples of correct and incorrect analysis are shown in Figure 6. The reasons of incorrect analysis are listed below.

- Inadequacy of the case frame dictionary IPAL regarding surface cases, distinction between obligatory case and optional case, and category specification for case slots (which means oversight of examples).
- Inadequacy of the thesaurus dictionary BGH. This problem is closely related to the method for correlating the level of most specific common layer with their similarity value. Generally speaking, however, BGH is not reliable enough to calculate an accurate similarity value between words.
- Insufficiency of examples in the case frame dictionary. Some case slots have only one or two examples. This problem can be solved simply by adding the wrongly analyzed sentences as new examples of their proper case frames.

In order to see how well these test sentences are analyzed by conventional SM methods, we translated test sentences of type 3 and 4, which have structural ambiguity, by two commercial machine translation (MT) systems. The commercial MT systems have a lot of heuristic rules, but they are thought to be based on tens or hundreds of SMs. We evaluated their outputs based on whether the syntactic analysis of Japanese sentences is correct or not (in the right part of Table 4). Furthermore, we compared the analysis results by our method with those by commercial MT systems (Table 3). We can see that the disambiguation of such complex sentences are fairly difficult for conventional SM methods and that our example-based method is significantly better at structural disambiguation than conventional SM methods.

	Our method			MT system I			MT system II		
	correct	incorrect	sr*	correct	incorrect	sr*	correct	incorrect	sr*
Type 1	219	0	100%	-	-	-	-	-	-
Type 2	12	0	100%	-	-	-	-	-	-
Type 3	183	13	93%	133	63	68%	147	49	75%
Type 4	33	4	89%	12	25	32%	12	25	32%
Type 3 and 4	216	17	93%	145	88	62%	159	74	68%

*success ratio

Table 4: Results of experiments.

5 Concluding Remarks

We have proposed a method that detects the case structure not only for a simple sentence, but also for a compound or a complex sentence. In this method, word ambiguity for verbs and structural ambiguity are solved simultaneously. The basis of this method is the process of selecting a proper case frame for the input sentence by matching it with example sentences in the case frame dictionary. We have reported experiments showing this method's superiority over the conventional, coarse-grained SM method.

The remaining problems are:

- In this paper we have hardly discussed the concept of thesaurus. Not only case structure analysis but also many other kinds of natural language processing depend on the accuracy of the thesaurus employed. We need to do research on the framework of a thesaurus where the relations of words are handled in various aspects and also research on a method for automatic construction of such a thesaurus.

- At present we first detect the scope of the coordinate structure and then detect the case structure of a sentence. However, it is desirable that the coordinate structure and the case structure of a sentence are evaluated by one combined measure as a whole. In order to do this without the combinatorial explosion of ambiguities, we need to devise a data structure and a search method for handling these problems together, or need to devise a method for judging dynamically which information is the most reliable.

References

[1] Katz, J. — Fodor, J. (1963) "The structure of a semantic theory". In: *Language* 39, 170-210.

[2] Longman Group Ltd. (1987) *Longman Dictionary of Contemporary English.*

[3] Nagao, M et al. (1985) "Outline of Machine Translation Project of the Science and Technology Agency". In: *J.IPS Japan* Vol.26, No.10 (in Japanese).

[4] Information-technology Promotion Agency, Japan (1987) *IPA Lexicon of the Japanese Language for computers IPAL (Basic Verbs).* (in Japanese).

[5] Ikehara, S et al. (1991) "Semantic Analysis Dictionaries for Machine Translation". In: *IPSJ-NLP* 84-13 (in Japanese).

[6] Kurohashi, S — Nagao, M (1992) "Dynamic Programming Method for Analyzing Conjunctive Structures in Japanese". In: *Proc. of the 14th International Conference on Computational Linguistics.*

[7] Nagao, M (1992) "Some Rationales and Methodologies for Example-based Approach". In: *Proc. of Workshop on Future Generation Natural Language Processing.* UMIST, Manchester.

[8] Inagaki, H et al. (1988) "Modification Analysis using Semantic Pattern". In: *IPSJ-NLP* 67-5 (in Japanese).

[9] Nagao, K. (1990) "Dependency Analyzer : A Knowledge-Based Approach to Structural Disambiguation". In: *Proc. of the 13th International Conference on Computational Linguistics.*

[10] National Language Research Institute (1964) *Word List by Semantic Principles.* Syuei Syuppan (in Japanese).

GLR* – An Efficient Noise-skipping Parsing Algorithm For Context Free Grammars

Alon Lavie and Masaru Tomita

School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
email: lavie@cs.cmu.edu

Abstract

This paper describes GLR*, a parser that can parse *any* input sentence by ignoring unrecognizable parts of the sentence. In case the standard parsing procedure fails to parse an input sentence, the parser nondeterministically skips some word(s) in the sentence, and returns the parse with fewest skipped words. Therefore, the parser will return some parse(s) with any input sentence, unless no part of the sentence can be recognized at all.

The problem can be defined in the following way: Given a context-free grammar G and a sentence S, find and parse S' - the largest subset of words of S, such that $S' \in L(G)$.

The algorithm described in this paper is a modification of the Generalized LR (Tomita) parsing algorithm [Tomita, 1986]. The parser accommodates the skipping of words by allowing shift operations to be performed from inactive state nodes of the Graph Structured Stack. A heuristic similar to beam search makes the algorithm computationally tractable.

There have been several other approaches to the problem of robust parsing, most of which are special purpose algorithms [Carbonell and Hayes, 1984], [Ward, 1991] and others. Because our approach is a modification to a standard context-free parsing algorithm, all the techniques and grammars developed for the standard parser can be applied as they are. Also, in case the input sentence is by itself grammatical, our parser behaves exactly as the standard GLR parser.

The modified parser, GLR*, has been implemented and integrated with the latest version of the Generalized LR Parser/Compiler [Tomita *et al.*, 1988], [Tomita, 1990].

We discuss an application of the GLR* parser to spontaneous speech understanding and present some preliminary tests on the utility of the GLR* parser in such settings.

1 Introduction

In this paper, we introduce a technique for substantially increasing the robustness of syntactic parsers to two particular types of extra- grammaticality: noise in the input, and limited grammar coverage. Both phenomena cause a common situation, where the input contains words or fragments that are unparsable. The distinction between these two types of extra- grammaticality is based to a large extent upon whether or not the unparsable fragment, in its context, can be considered grammatical by a linguistic judgment. This distinction may indeed be vague at times, and practically unimportant.

Our approach to the problem is to enable the parser to overcome these forms of extra-grammaticality by ignoring the unparsable words and fragments and focusing on the maximal subset of the input that is covered by the grammar. Although presented and implemented as an enhancement to the Generalized LR parsing paradigm, our technique is applicable in general to most phrase- structured based parsing formalisms. However, the efficiency of our parser is due in part to several particular properties of GLR parsing, and may thus not be easily trans-

ferred to other syntactic parsing formalisms.

The problem can be formalized in the following way: Given a context-free grammar G and a sentence S, find and parse S' - the largest subset of words of S, such that $S' \in L(G)$.

A naive approach to this problem is to exhaustively list and attempt to parse all possible subsets of the input string. The largest subset can then be selected from among the subsets that are found to be parsable. This algorithm is clearly computationally infeasible, since the number of subsets is exponential in the length of the input string. We thus devise an efficient method for accomplishing the same task, and pair it with an efficient search approximation heuristic that maintains runtime feasibility.

The algorithm described in this paper, which we have named GLR*, is a modification of the Generalized LR (Tomita) parsing algorithm. It has been implemented and integrated with the latest version of the GLR Parser/Compiler [Tomita *et al.*, 1988], [Tomita, 1990].

There have been several other approaches to the problems of robust parsing, most of which have been special purpose algorithms. Some of these approaches have abandoned syntax as a major tool in handling extra-grammaticalities and have focused on domain dependent semantic methods [Carbonell and Hayes, 1984], [Ward, 1991]. Other systems have constructed grammar and domain dependent fall-back components to handle extra-grammatical input that causes the main parser to fail [Stallard and Bobrow, 1992], [Seneff, 1992].

Our approach can be viewed as an attempt to extract from the input the maximal syntactic structure that is possible, within a purely syntactic and domain independent setting. Because the GLR* parsing algorithm is an enhancement to the standard GLR context-free parsing algorithm, all of the techniques and grammars developed for the standard parser can be applied as they are. In particular, the standard LR parsing tables are compiled in advance from the grammar and used "as is" by the parser in runtime. The GLR* parser inherits the benefits of the original parser in terms of ease of grammar development, and, to a large extent, efficiency properties of the parser itself. In the case that the input sentence is by itself grammatical, GLR* behaves exactly as the standard GLR parser.

The remaining sections of the paper are organized in the following way: Section 2 presents an outline of the basic GLR* algorithm itself, followed by a detailed example of the operation of the parser on a simple input string. In section 3 we discuss the search heuristic that is added to the basic GLR* algorithm, in order to ensure its runtime feasibility. We discuss an application of the GLR* algorithm to spontaneous speech understanding, and present some preliminary test results in section 4. Finally, our conclusions and further research directions are presented in section 5.

(1) S → NP VP
(2) NP → det n
(3) NP → n
(4) NP → NP PP
(5) VP → v NP
(6) PP → p NP

Figure 1: A Simple Natural Language Grammar

2 The GLR* Parsing Algorithm

The GLR* parsing algorithm is an extension of the Generalized LR Parser, as implemented in the Universal Parser Architecture developed at CMU [Tomita, 1986]. This implementation incorporates an SLR(0) parsing table.

The parser accommodates skipping words of

the input string by allowing shift operations to be performed from inactive state nodes in the Graph Structured Stack (GSS). Shifting an input symbol from an inactive state is equivalent to skipping the words of the input that were encountered after the parser reached the inactive state and prior to the current word being shifted. Since the parser is LR(0), reduce operations need not be repeated for skipped words (the reductions do not depend on any lookahead). Information about skipped words is maintained in the symbol nodes that represent parse sub-trees.

An initial version of the GLR* parser has been implemented in Lucid Common Lisp, in the integrated environment of the Universal Parser Architecture.

	Reduce	Shift					Goto			
State		det	n	v	p	$	NP	VP	PP	S
0		sh3	sh4				2			1
1						acc				
2				sh7	sh8			5	6	
3			sh9							
4	r3									
5	r1									
6	r4									
7		sh3	sh4				10			
8		sh3	sh4				11			
9	r2									
10	r5				sh8				6	
11	r6				sh8				6	

Table 1: SLR(0) Parsing Table for Grammar in Figure 1

2.1 An Example

To clarify how the proposed GLR* parser actually works, in lieu of a more formal description of the algorithm itself, we present a step by step runtime example. For the purpose of the example, we use a simple natural language grammar that is shown in Figure 1. The terminal symbols of the grammar are depicted in lower-case, while the non-terminals are in upper-case. The grammar is compiled into an SLR(0) parsing table, which is displayed in Table 1. Note that since the table is SLR(0), the reduce actions are independent of any lookahead. The actions on states 10 and 11 include both a shift and a reduce.

To understand the operation of the parser, we now follow some steps of the GLR* parsing algorithm on the input `x = det n v n det p n`. This input is ungrammatical due to the second "`det`" token. The maximal parsable subset of the input in this case is the string that includes all words other than the above mentioned "`det`".

In the figures ahead, which graphically display the GSS of the parser in various stages of the parsing process, we use the following notation:

- An *active* (top level) state node is represented by the symbol "`@`", with the state number indicated above it. Actions that are attached to the node are indicated to the right of the node.
- An *inactive* state node is represented by the symbol "`*`". The state number is indicated above the node and actions that are attached to the node are indicated above the state number.
- Grammar symbol nodes are represented by the symbol "`#`", with the grammar symbol itself displayed above it.

```
0                                  after initialization
@ sh3                              (and empty reduce phase)
```

Figure 2: Initial GSS

```
sh4                                after first shift phase
 0  det  3                         (and empty reduce phase)
 *---#---@ sh9
```

Figure 3: GSS after first shift phase

The parser operates in phases of shifts and reductions. We follow the GSS of the parser following each of these phases, while processing the input string. Reduce actions are distributed to the active nodes during initialization and after each shift phase. Shift actions are distributed after each reduce phase. Note that the GLR* parsing algorithm distributes shift actions to *all* state nodes (both active and inactive), whereas the original parser distributed shift actions only to active nodes. Reduce actions are distributed only to active state nodes.

Figure 2 is the initial GSS, with an active state node of state 0. Since there are no reduce actions from state 0, the first reduce phase is empty. With the first input token being "det", the shift action attached to state node 0 is "sh3".

Figure 3 shows the GSS after the first shift phase. The symbol node labeled "det" has been shifted and connected to the initial state node and to the new active state node of state 3. Since there are no reduce actions from state 3, the next reduce phase is empty. The next input token is "n". Shift actions are distributed by the algorithm to both the active node of state 3 and the inactive node of state 0, as can be seen in Figure 3.

Figure 4 shows the GSS after the next shift phase. The input token "n" was shifted from both state nodes, creating active state nodes of states 9 and 4. The shifting of the input token "n" from state 0 corresponds to a parsing possibility in which the first input token "det" is skipped. Reduce actions are distributed to both of the active nodes.

The following reduce phase reduces both branches into noun phrases. The two "NP"s are packed together by a local ambiguity packing procedure. Using information on skipped words that is maintained within the symbol nodes, the ambiguity packing can detect that one of the noun phrases (the one that was reduced from "det n") is more complete, and the other noun phrase is discarded. The resulting GSS is displayed in Figure 5. Shift actions with the next input token "v" are then distributed to all the state nodes. However, in this case, only state 2 allows a shift of "v" into state 7.

Figure 6 shows the GSS after the third shift phase. The state 7 node is the only active node at this point. Since no reduce actions are specified for this state, the fourth reduce phase is empty. Shift actions with the next input token "n" are distributed to all state nodes, as can be seen in the figure.

Figure 7 shows the GSS after the fourth shift phase and Figure 8 after the fifth reduce phase. Note that there are no active state nodes after the fifth reduce phase. This is due to the fact that none of the state nodes produced by the reduce phase allow the shifting of the next input token "det". The original parser would have thus failed as this point. However, the GLR* parser succeeds in distributing shift actions to two inactive state nodes in this case.

For the sake of brevity we do not continue to further follow the parsing step by step. The final GSS is displayed in Figure 9. Several different parses, with different subsets of skipped words are actually packed into the single "S" node seen at the bottom of the figure. The parse that corre-

sponds to the maximal subset of the input is the one in which the second "det" is the only word skipped.

Figure 4: GSS after second shift phase

2.2 Efficiency of the Parser

Efficiency of the parser is achieved by a number of different techniques. The most important of these is a sophisticated process of local ambiguity packing and pruning. A local ambiguity is a part of the input sentence that corresponds to a phrase (thus, reducible to some non-terminal symbol of the grammar), and is parsable in more than one way. The process of skipping words creates a large number of local ambiguities. For example, the grammar in Figure 1 allows both determined and undetermined noun phrases (rules 2 and 3). As seen in the example presented earlier, this results in the creation of two different noun phrase symbol nodes for the initial fragment "det n". The first node is created for the full phrase after a reduction according to the first rule. A second symbol node is created when the determiner is skipped and a reduction by the second rule takes place.

Locally ambiguous symbol nodes are detected as nodes that are surrounded by common state nodes in the GSS. The original GLR parser detects such local ambiguities and packs them into a single symbol node. This procedure was extended in the GLR* parser. Locally ambiguous symbol nodes are compared in terms of the words skipped within them. In cases such as the example described above, where one phrase has more skipped words than the other, the phrase with more skipped words is discarded in favor of the more complete parsed phrase. This subsuming operation drastically reduces the number of parses being pursued by the parser.

Another technique employed to increase the efficiency of the parser is the merging of state nodes of the same state after a reduce phase and after a shift phase. This allows the parsing through the GSS to continue with fewer state nodes.

2.3 Selecting the Best Maximal Parse

An obvious and unsurprising side effect of the GLR* parser is an explosion in the number of parses found by the parser. In principle, we are only interested in finding the maximal parsable subset of the input string (and its parse). However, in many cases there are several distinct maximal parses, each consisting of a different subset

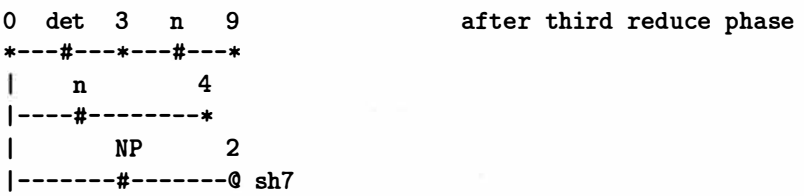


Figure 5: GSS after third reduce phase

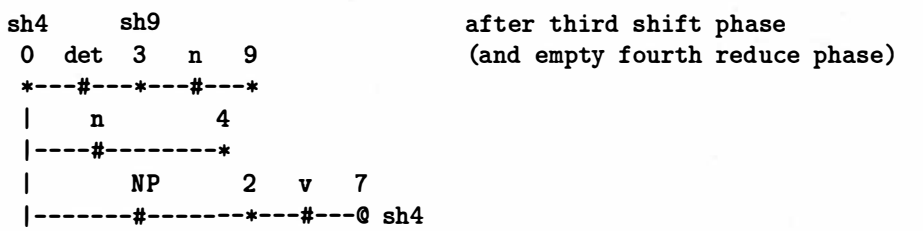


Figure 6: GSS after third shift phase

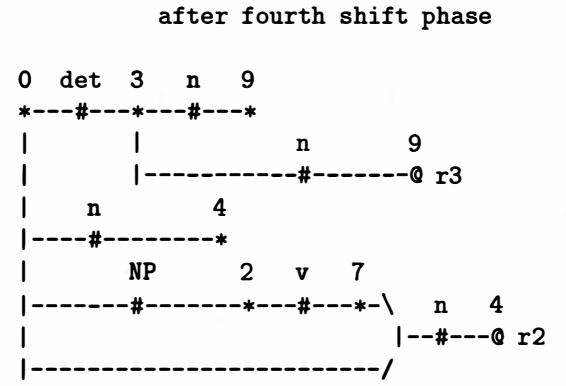


Figure 7: GSS after fourth shift phase

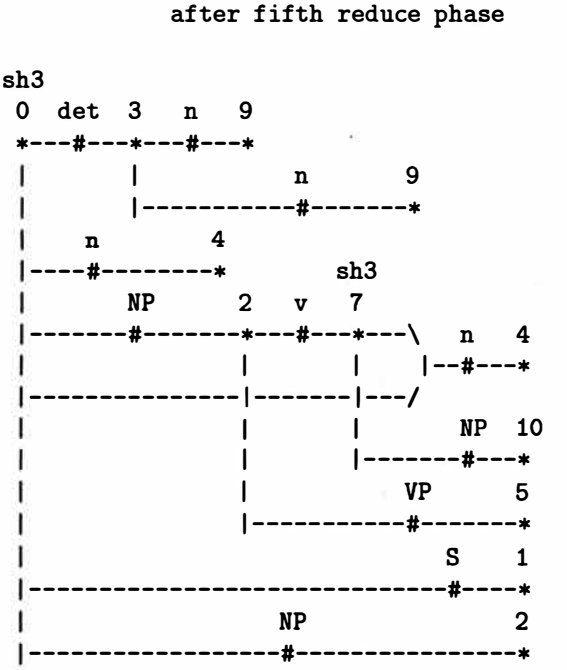


Figure 8: GSS after fifth reduce phase

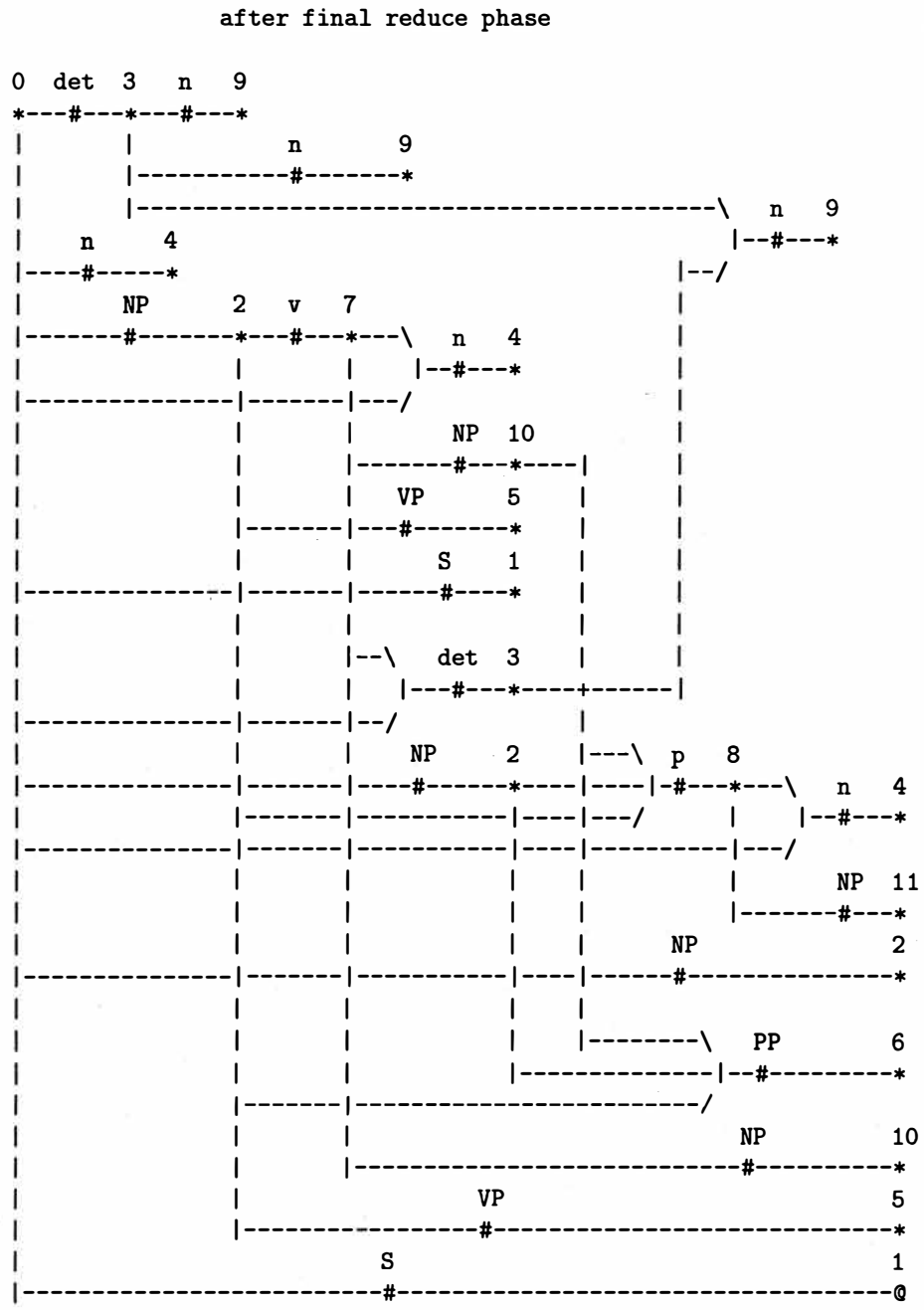


Figure 9: GSS after final reduce phase

of words of the original sentence. Additionally, there are cases where a parse that is not maximal in terms of the number of words skipped may be deemed preferable.

To select the "best" parse from the set of parses returned by the parser, we use a scoring procedure that ranks each of the parses found. We then select the parse that was ranked best. [1] Presently, our scoring procedure is rather simple. It takes into account the number of words skipped and the fragmentation of the parse (i.e. the number of S-nodes that the parsed input sentence was divided into). Both measures are weighed equally. Thus a parse that skipped one word but parsed the remaining input as a single sentence is preferred over a parse that fragments the input into three sentences, without skipping any input word.

On the top of our current research goals is the enhancement of this simple scoring mechanism. We plan on adding to our scoring function several additional heuristic measures that reflect various syntactic and semantic properties of the parse tree. We will measure the effectiveness of our enhanced scoring function in ranking the parse results by their desirability.

3 The Beam Search Heuristic

Although implemented efficiently, the basic GLR* parser is still not guaranteed to have a feasible running time. The basic GLR* algorithm described computes parses of all parsable subsets of the original input string, the number of which is potentially exponential in the length of the input string. Our goal is to find parses of maximal subsets of the input string (or almost maximal subsets). We have therefore developed and added to the parser a heuristic that prunes parsing options that are not likely to produce a maximal parse. This process has been traditionally called "beam search".

A direct way of adding a beam search to the parser would be to limit the number of active state nodes pursued by the parser at each stage, and continue processing only active nodes that are most promising in terms of the number of skipped words associated with them. However, the structure of the GSS makes it difficult to associate information on skipped words directly with the state nodes. [2] We have therefore opted to implement a somewhat different heuristic that has a similar effect.

Since the skipping of words is the result of performing shift operations from inactive state nodes of the GSS, our heuristic limits the number of inactive state nodes from which a input symbol is shifted. At each shift stage, shift actions are first distributed to the active state nodes of the GSS. This corresponds to no additional skipped words at this stage. If the number of state nodes that allow a shift operation at this point is less than a predetermined constant threshold (the "beam-limit"), then shift operations from inactive state nodes are also considered. Inactive states are processed in an ordered fashion, so that shifting from a more recent state node that will result in fewer skipped words is considered first. Shift operations are distributed to inactive state nodes in this way until the number of shifts distributed reaches the threshold.

This beam search heuristic reduces the runtime of the GLR* parser to within a constant factor of the original GLR parser. Although it is not guaranteed to find the desired maximal parsable subset of the input string, our preliminary tests have shown that it works well in practice.

The threshold (beam-limit) itself is a parameter that can be dynamically set to any constant value at runtime. Setting the beam-limit to a value of 0 disallows shifting from inactive states all together, which is equivalent to the original GLR parser. In preliminary experiments that we have conducted (see next section) we have achieved good results with a setting of the beam-limit to values in the range of 5 to 10. There exists a direct tradeoff between the value of the beam-limit and the runtime of the GLR* parser. With a set value of 5, our tests have indicated a runtime that is within a factor of 2-3 times that of the original GLR parser, which amounts to a parse time of only several seconds on sentences that are up to 30 words long.

[1] The system will display the n best parses found, where the parameter n is controlled by the user at runtime. By default, we set n to one, and the highest ranking parse is displayed.

[2] This is due to the fact that state nodes are merged, so that a state node may be common to several different parses, with different skipped words associated with each parse.

	Robust Parser
	number (and percent)
Parsable	99
Unparsable	1
Good/Close Parses	77
Bad Parses	22

Table 2: Performance of the GLR* Parser on Spontaneous Speech

4 Parsing of Spontaneous Speech Using GLR*

4.1 The Problem of Parsing Spontaneous Speech

As a form of input, spontaneous speech is full of noise and irrelevances that surround the meaningful words of the utterance. Some types of noise can be detected and filtered out by speech recognizers that process the speech signal. A parser that is designed to successfully process speech recognized input must however be robust to various forms of noise, and be able to weed out the meaningful words from the rest of the utterance.

When parsing spontaneous spoken input that was recognized by a speech recognition system, the parser must deal with three major types of extra-grammaticality:

- Noise due to the spontaneity of the speaker, such as repeated words, false beginnings, stuttering, and filled pauses (i.e. "ah", "um", etc.).
- Ungrammaticality that is due to the language of the speaker, or to the coverage of the grammar.
- Noise due to errors of the speech recognizer.

We have conducted two preliminary experiments to evaluate the GLR* parser's ability to overcome the first two types of extra-grammaticality. We are in the process of experimenting with the GLR* parser on actual speech recognized output, in order to test its capabilities in handling errors produced by the speech recognizer.

4.2 Parsing of Noisy Spontaneous Speech

The first test we conducted was intended to evaluate the performance of the GLR* parser on noisy sentences typical of spontaneous speech. The parser was tested on a set of 100 sentences of transcribed spontaneous speech dialogues on a conference registration domain. The input is hand-coded transcribed text, not processed through any speech recognizer. The grammar used was an upgraded version of a grammar for the conference registration task, developed and used by the JANUS speech-to-speech translation project at CMU [Waibel et al. 1991]. Since the test sentences were drawn from actual speech transcriptions, they were not guaranteed to be covered by the grammar. However, since the test was meant to focus on spontaneous noise, sentences that included verbs and nouns that were beyond the vocabulary of the system were avoided. Also pruned out of the test set were short opening and closing sentences (such as "hello" and "goodbye"). The transcriptions include a multitude of noise in the input. The following example is one of the sentences from this test set:

```
"fckn2_10 /ls/ /h#/ um okay {comma}
then yeah I am disappointed {comma}
*pause* but uh that is okay {period}"
```

The performance results are presented in Table 2. Note that due to the noise contaminating the input, the original parser is unable to parse a single one of the sentences in this test set. The GLR* parser succeeded to return some parse result in all but one of the test sentences. However, since returning a parse result does not by itself guarantee an analysis that adequately reflects the meaning of the original utterance, we reviewed the parse results by hand, and classified them into the categories of "good/close" and "bad" parses. The results of this classification are included in the table.

4.3 Grammar Coverage

We conducted a second experiment aimed exclusively on evaluating the ability of the GLR* parser to overcome limited grammar coverage. In this experiment, we compared the results of the GLR* parser with those of the original GLR parser on a common set of sentences using the same grammar. We used the grammar from the spontaneous speech experiment for this test as well. The common test set was a set of 117 sentences from the conference registration task of the JANUS project. These sentences are simple synthesized text sentences. They contain no spontaneous speech noise, and are not the result of any speech recognition processing. Once again, to evaluate the quality of the parse results returned by the parser, we classified the parse results of both parsers by hand into two categories: "good/close parses" and "bad parses". The results of the experiment are presented in Table 3.

The results indicate that using the GLR* parser results in a significant improvement in performance. The percentage of sentences, for which the parser returned good or close parses increased from 52% to 70%, an increase of 18%. Fully 97% of the test sentences (all but 3) are parsable by the GLR* parser, an increase of 36% over the original parser. However, this includes a significant increase (from 9% to 27%) in the number of bad parses found. Thus, fully half of the additional parsable sentences of the set return with parses that may be deemed bad.

The results of the two experiments clearly point to the following problem: Compared with the GLR* parser, the original GLR parser, although fragile, returned results of relatively good quality, when it succeeded in parsing the input. The GLR* parser, on the other hand, will succeed in parsing almost any input, but this parse result may be of little or no value in a significant portion of cases. This indicates a strong need in the development of methods for discriminating between good and bad parse results. We intend to try and develop some effective heuristics to deal with this problem. The problem is also due in part to the ineffectiveness of the simple heuristics currently employed for selecting the best parse result from among the large set of parses returned by the parser. As mentioned earlier, we intend to concentrate efforts on developing more sophisticated and effective heuristics for selecting the best parse.

5 Conclusions and Future Research Directions

Motivated by the difficulties that standard syntactic parses have in dealing with extra-grammaticalities, we have developed GLR*, an enhanced version of the standard Generalized LR parser, that can effectively handle two particular problems that are typical of parsing spontaneous speech: noise contamination and limited grammar coverage.

Given a grammar G and and input string S, GLR* finds and parses S', the maximal subset of words of S, such that S' is in the language $L(G)$. The parsing algorithm accommodates the skipping of words and fragments of the input string by allowing shift operations to be performed from inactive states of the GSS (as well as from the active states, as is done by the standard parser). The algorithm is coupled with a beam-search-like heuristic, that controls the process of shifting from inactive states to a limited beam, and

	Original Parser		Robust Parser	
	number	percent	number	percent
Parsable	71	61%	114	97%
Unparsable	46	39%	3	3%
Good/Close Parses	61	52%	82	70%
Bad Parses	10	9%	32	27%

Table 3: Performance of the GLR* Parser vs. the Original Parser

maintains computational tractability.

Most other approaches to robust parsing have suffered to some extent from a lack of generality and from being domain dependent. Our approach, although limited to handling only certain types of extra-grammaticality, is general and domain independent. It attempts to maximize the robustness of the parser within a purely syntactic setting. Because the GLR* parsing algorithm is a modification of the standard GLR context-free parsing algorithm, all of the techniques and grammars developed for the standard parser can be applied as they are. In the case that the input sentence is by itself grammatical, GLR* behaves exactly as the standard GLR parser. The techniques used in the enhancement of the standard GLR parser into the robust GLR* parser are in principle applicable to other phrase-structure based parsers.

Preliminary experiments conducted on the effectiveness of the GLR* parser in handling noise contamination and limited grammar coverage have produced encouraging results. However, they have also pointed out a definite need to develop effective heuristics that can select the best parse result from a potentially large set of possibilities produced by the parser. Since the GLR* parser is likely to succeed in producing some parse in practically all cases, successful parsing by itself can no longer be an indicator to the value and quality of the parse result. Thus, additional heuristics need to be developed for evaluating the quality of the parse found.

We intend to concentrate on developing such effective heuristics that will complement the GLR* parser, and boost its performance in handling spontaneously spoken input. We plan to conduct extensive experiments with speech recognized input to evaluate our system and guide its further development. We also plan to investigate the potential of the GLR* parser in several other application areas and domains.

References

[Carbonell and Hayes, 1984] J. G. Carbonell and P. J. Hayes. Recovery Strategies for Parsing Extragrammatical Language. *Technical Report CMU-CS-84-107*, 1984.

[Seneff, 1992] S. Seneff. A relaxation method for understanding spontaneous speech uttera nces. In *Proceedings of DARPA Speech and Natural Language Workshop*, pages 299–304, February 1992.

[Stallard and Bobrow, 1992] D. Stallard and R. Bobrow. Fragment processing in the DELPHI system. In *Proceedings of DARPA Speech and Natural Language Workshop*, pages 305–310, February 1992.

[Tomita *et al.*, 1988] M. Tomita, T. Mitamura, H. Musha, and M. Kee. The Generalized LR Parser/Compiler - Version 8.1: User's Guide . *Technical Report CMU-CMT-88-MEMO*, 1988.

[Tomita, 1986] M. Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Hingham, Ma., 1986.

[Tomita, 1990] M. Tomita. The Generalized LR Parser/Compiler - Version 8.4. In *Proceedings of International Conference on Computational Lingui stics (COLING-90)*, pages 59–63, Helsinki, Finland, 1990.

[Ward, 1991] W. Ward. Understanding spontaneous speech: The Phoenix system. In *Proceedings of IEEE International Conference on Acoustics, Spee ch and Signal Processing (ICASSP)*, pages 365–367, April 1991.

The Use of Bunch Notation in Parsing Theory

René Leermakers

Institute for Perception Research
P.O. Box 513, 5600 MB EINDHOVEN
email: leermake@prl.philips.nl

Abstract

Much of mathematics, and therefore much of computer science, is built on the notion of sets. In this paper it is argued that in parsing theory it is sometimes convenient to replace sets by a related notion, *bunches*. The replacement is not so much a matter of principle, but helps to create a more concise theory. Advantages of the bunch concept are illustrated by using it in descriptions of a formal semantics for context-free grammars and of functional parsing algorithms.

1 Introduction

The semantics of a context-free grammar can be given in a number of ways. In the three most important interpretations, a grammar is viewed as a rewriting system, or as a set of inequalities, or as an abstract program. The latter two interpretations are discussed in this paper, using a variant of set notation, called *bunch notation*. Subsequently, a new Earley-like recursive-ascent parser is formulated with the same notation.

There are two major differences between sets and bunches. One difference is that a bunch with one element is identified with that one element (the singleton property). Moreover, a function or operator that is defined on some domain may be applied to a bunch of elements that belong to that domain. Such an application, say $f(X)$, causes the function f to be applied to each separate element of the bunch X, after which the results are combined in a bunch, which is the result of $f(X)$. This is called the distributivity property of bunches. To define bunch-valued expressions, a variant of the notation of (Norvell — Hehner, 1992) is used.

A language can be defined as a bunch of strings rather than as a set of strings. Then, both the singleton and distributivity properties of bunches simplify the formalization of the natural interpretation of context-free grammars, in which grammar rules are seen as constraints on the possible assignments of languages to nonterminals.

Multiple-valued (recursive descent and ascent) parsing functions can be defined as bunch-valued functions. Again, both the singleton and distributivity properties have advantages. The singleton property smoothes the transition from parsing algorithms for general grammars to deterministic algorithms for LL(k) and LALR(k) grammars. The distributivity property makes it possible to write succinct formulae in bunch notation, that 'blow up' if translated into set notation. Finally, Norvell — Hehner's (1992) bunch notation has an advantage over standard set notation when it comes to defining functional algorithms, in that it resembles traditional notation for specifying programs. In particular, the definition of recursive parsing algorithms is in terms of a construct akin to Dijkstra's guarded commands (Dijkstra, 1976).

The paper starts with an introduction to bunch notation. The first application of the notation is a reformulation of the natural semantics of context-free grammars. Subsequent sections give functional definitions of known recursive descent algorithms and a new recursive ascent recognition algorithm.

2 Bunch notation

In standard mathematics, a (total) function $f : \mathcal{A} \mapsto \mathcal{B}$ associates one element of $\mathcal{B}$ with each element of $\mathcal{A}$. A function is a special case of a relation, which may associate any number of elements of $\mathcal{B}$ with each element of $\mathcal{A}$. Conversely, each relation can be seen as a special kind of a function too: a set-valued function that, if applied to $a \in \mathcal{A}$, yields the set of elements of $\mathcal{B}$ associated with a by the relation. Alternatively, a relation may be viewed as a nondeterministic function: of all elements of $\mathcal{B}$ related to some element $a \in \mathcal{A}$, the nondeterministic function arbitrarily picks one.

The set-valuedness of functions associated with a relation has one peculiar consequence. Take a function $f : \mathcal{A} \mapsto \mathcal{B}$, and view it as a relation with the special property that it relates only one element of $\mathcal{B}$ to each element of $\mathcal{A}$. Next, use the mapping from relations to set-valued or nondeterministic functions to view the relation as a function again. Then one would expect to re-obtain the original function f. If the relation is mapped to a nondeterministic function, this is indeed the case: the function happens to be deterministic and is equal to f. Using standard sets, however, the set-valued function associated with the relation associated with f, produces a set with exactly one element (a singleton) where f produces that element. This suggests that it is better not to see relations as set-valued functions, but rather as *bunch-valued* functions. A bunch is a set with some non-standard properties, so that it can be interpreted in an alternative way: a bunch is also a process that nondeterministically produces one of its values. The alternative interpretation implies the following three properties of bunches:

1. The process that corresponds to a bunch with one value (a singleton) is deterministic: it can only produce that one value. Therefore: a singleton is identified with its only element.

2. The process that corresponds to a bunch produces definite values. Therefore: elements of bunches can not be bunches with cardinality $\neq 1$.

3. If f is a function and x is a bunch that can produce the values $e_1 ... e_k$, then $f(x)$ can take the values $f(e_1) ... f(e_k)$. Therefore: functions distribute over bunches.

With these properties, a bunch simultaneously allows two interpretations. In the set-valued interpretation it is just a collection of values. In the nondeterministic interpretation one value is randomly taken out of this collection.

Bunches are the result of bunch expressions. Given two bunch expressions x and y, their bunch union $x|y$ denotes a process that could either produce a value of x or a value of y. Bunch union has the same properties as set union: it is associative, commutative and idempotent. The main difference with sets is that a bunch is not 'one thing' if it has more than one element. This is why a bunch with many elements cannot be one element of another bunch: it can only be many elements of another bunch. This is also why a bunch with many elements cannot be passed to a function or operator as one thing. Here are a few examples of equalities for bunch expressions that illustrate the above:

$$3 + (1|2) \equiv 4|5$$
$$(3|4) + (1|2) \equiv 4|5|5|6 \equiv 4|5|6$$
$$cos(\pi|0) \equiv -1|1$$
$$(1|2) > 3 \equiv \mathbf{false}|\mathbf{false} \equiv \mathbf{false}$$

If e is one of the values a bunch expression x can take, we write $e \leftarrow x$. Here and henceforth, e is a definite value or, what is the same, a singleton bunch. As definite values are also bunches, and elements of bunches cannot be bunches with cardinality unequal to one, the distinction between $\in$ and $\subseteq$ is no longer needed: if, for all e, $e \leftarrow x$ implies $e \leftarrow y$ then we write $x \leftarrow y$. In words, x is a sub-bunch of y, or, x is smaller than y.

Bunch expressions can be simple or complex. The simplest simple bunch expression is the empty bunch *null*. It is the identity of bunch union. Other simple bunch expressions are enumerations. The following is a formal definition of simple bunch expressions with elements from a (possibly infinite) set of definite values:

1. *null* is a simple bunch expression;

2. if e is a definite value then e is a simple bunch expression;

3. if x and y are simple bunch expressions then $x|y$ is a simple bunch expression.

A simple bunch expression may be rewritten into an equivalent simple bunch expression using $(x|y)|z \equiv x|(y|z) \equiv x|y|z$, $x|y \equiv y|x$, $x|x \equiv x$ and $x|null \equiv x$. Assuming some ordering on the set of definite values, it is not difficult to define a canonical form for each simple bunch expression, which may serve as a unique representation of the bunch denoted. The bunch expression *all* denotes the smallest bunch such that $e \leftarrow all$ for all definite values e.

Complex bunch expressions are constructed with variables. Unless stated otherwise, variables have types that consist of definite values only. Such variables are called definite; they cannot be bound to bunches with cardinality unequal to one. Given a proposition P and bunch expressions x and y, the expression

$$\textbf{if } P \textbf{ then } x \textbf{ else } y \qquad (1)$$

is a complex bunch expression. It contains free variables if P, x, or y contain free variables (nontrivial P do). It will be clear that for each assignment of values to the variables the complex expression (1) is equivalent to x if P is true and to y otherwise. Free variables in bunch expressions can be bound by λ-abstraction. If i is a variable and x is a bunch expression, then

$$\lambda i \cdot x$$

is a bunch-valued function. For any definite value e,

$$\lambda i \cdot x(e) \stackrel{def}{\equiv} \text{Substitute } e \text{ for free variables } i \text{ everywhere in } x.$$

This definition holds only for *definite* values e. Note, therefore, that it is important to distinguish between functions and expressions. In expression x in $\lambda i \cdot x$, variable i may occur more than once. If function $\lambda i \cdot x$ is applied to a bunch y, then the distributivity of functions over bunches means that the function applies to each $e \leftarrow y$ separately. That is, if $x = mult(i, i)$ then $\lambda i \cdot x(2|3) = mult(2,2)|mult(3,3)$; $mult(2,3)$ is not included. In jargon, our functional language is characterized as having a semantics such that functions are not *unfoldable*: a function invocation cannot be textually replaced by the expression that defines the function, if function parameters are not definite (Sondergard — Sesoft, 1990).

This is practically all we need to know about bunches. Let us just add some notations:

$$P \rhd x \stackrel{def}{\equiv} \textbf{if } P \textbf{ then } x \textbf{ else } null$$

$$\textbf{let } i \cdot x \stackrel{def}{\equiv} \lambda i \cdot x(all)$$

The bunch *all* will in general be infinite, so that a function that distributes over it might produce an infinite bunch as well. In our application, however, the structure of bunch expressions will be such that **let**'s produce only finite bunches.

The following laws are useful for manipulating bunch expressions:

$$(P_1 \wedge P_2) \rhd x \equiv P_1 \rhd (P_2 \rhd x), \qquad (2)$$

$$(P_1 \vee P_2) \rhd x \equiv (P_1 \rhd x)|(P_2 \rhd x), \qquad (3)$$

$$\textbf{let } i \cdot (i \leftarrow x \rhd f(i)) \equiv f(x). \qquad (4)$$

In (4) it is assumed that i does not occur free in x. These laws are easy to prove: the first two follow from the definition of $\rhd$, and the third follows from the distributivity of function application over bunches.

Normally, set-valued functions are defined using set comprehension according to the schema

$$f(X) = \{A(X,Y) | \exists_Z P(X,Y,Z)\}, \qquad (5)$$

where P is a predicate, A is a function and X, Y, Z are variables or sets of variables. Now let us define a related bunch-valued function, called f_b:

$$f_b = \lambda X \cdot (\textbf{let} Y \cdot (\textbf{let} Z \cdot (P(X,Y,Z) \rhd A(X,Y)))) \qquad (6)$$

It follows that f and f_b are equivalent if the latter is interpreted as producing a set. A nice aspect of (6) is that its algorithmic content is more explicit than the algorithmic content of (5); because **let** is defined as a function application to *all*, it is explicit that (6) involves searching over all values of Y, Z.

In this paper we will use a notational convention that removes the λ's and the **let**'s from definitions such as (6). Instead of (6) we write

$$f_b(X) = \quad P(X,Y,Z) \rhd A(X,Y). \qquad (7)$$

So $\lambda X \cdot$ has changed into a formal argument on the left-hand side and we adopt the convention that free variables at the right-hand side (here Y,Z) are bound by **let**'s. The scope of such an implicit **let** is in practice always clear: it is from the first occurrence of the variable usually until "|", or else until the end of the bunch expression. Thus, whenever an expression $P \rhd x$ is encountered in this paper, with some free variables, its meaning is that all possible values of the free variables must be tried to make the guard P true and all results x must be combined in one bunch.

Lastly, note that (7) is equivalent to

$$f_b(X) = \exists_Z(P(X,Y,Z)) \rhd A(X,Y).$$

That is, both functions produce the same bunch for every X. The algorithmic interpretation of both formulae is not exactly the same, however (see below). Therefore, if a variable appears only in a guard, like Z in (7), it is implicitly subject to existential quantification.

Algorithmic interpretation

In what follows, bunch-valued functions are either known computable functions, or their definitions have the following general format:

$$\begin{aligned} f(X) = \; & P_1(X,Y_1) \rhd A_1(X,Y_1) \mid \\ & P_2(X,Y_2) \rhd A_2(X,Y_2) \mid \\ & \vdots \\ & P_k(X,Y_k) \rhd A_k(X,Y_k), \end{aligned}$$

where X is a collection of input parameters and Y_i are collections of variables subject to **let** quantification. P_i are predicates and A_i are bunch-valued functions. Both P_i and A_i may involve other applications of bunch-valued functions. The intention is that bunches are interpreted as collections: bunch-valued functions produce all their results simultaneously. Function f then has a simple algorithmic (imperative) interpretation, which makes use of a bunch-valued variable "result":

$f(X) =$ result:=*null*;
for all Y_1 such that $P_1(X,Y_1)$ do
result:=result | $A_1(X,Y_1)$
od;
for all Y_2 such that $P_2(X,Y_2)$ do
result:=result | $A_2(X,Y_2)$
od;
.
.
.
for all Y_k such that $P_k(X,Y_k)$ do
result:=result | $A_k(X,Y_k)$
od;
return result

The invocations $A_i(X,Y_i)$ and function applications inside P_i are to be computed in the same vein. In this algorithmic interpretation, a function may or may not terminate. If it does not terminate, $f(X)$ does not define an algorithm. This may happen if the definition of $f(X)$ is circular, i.e. if the computation of some P_i or A_i involves $f(X)$ again.

The above function f is *deterministic* if for each X at most one proposition $P_i(X,Y_i)$ can be true, for only one value of Y_i, and function A_i is deterministic.

3 The natural semantics of grammars

Within the family of rewriting systems, context-free grammars have a distinguishing property: they have a declarative meaning. This means that a grammar can be understood not only by producing a sample of trial sentences with it, but also by viewing it as a collection of static statements about the language to be defined. This is the underlying reason for their intelligibility and their usefulness. In the natural interpretation, grammar symbols are seen as variables over languages and grammar rules as stipulations of relations between these variables. A grammar, in this view, is analogous to a collection of arithmetic inequalities with variables. Take, for instance, the following inequalities:

$$k \geq l + 3, \; l \geq 5.$$

A formal interpretation of this is that there are two symbols k and l here, that there is some assignment function h from these symbols to numbers, and that the inequalities restrict the possible values of h, via

$$h(k) \geq h(l) + 3,\ h(l) \geq 5.$$

Of course, there are still many functions h that satisfy these constraints but there is one that assigns the smallest possible numbers to the symbols: $h(k) = 8, h(l) = 5$.

Notation

A context-free grammar is a four-tuple $G = (V_N, V_T, P, S)$, where S is the start symbol, V_N is the *bunch* of nonterminals, V_T is the bunch of terminals. Furthermore, $V = V_N|V_T$ is the bunch of grammar symbols. Relating to grammar symbols, the following typed variables are used: $x, y \leftarrow V_T$, $\xi, \eta, \rho, \zeta \leftarrow V_T^*$, $A, B \leftarrow V_N$, $X, Y \leftarrow V$, $\alpha, \beta, \gamma, \delta, \mu, \nu \leftarrow V^*$. Lastly, P is the collection of grammar rules. A grammar rule for nonterminal A, with right-hand side α, is denoted as $A \rightarrow \alpha$. If β can be derived from α in any number of steps, we write $\alpha \xrightarrow{*} \beta$.

Languages

A language is a bunch of strings of terminals, i.e. a subbunch of V_T^*. Concatenation is an operation that is defined for (pairs of) strings. Therefore, it distributes over languages L and M, if these are concatenated:

$$LM = \ \xi \leftarrow L \wedge \rho \leftarrow M \ \triangleright \xi\rho. \qquad (8)$$

This equation is referred to as the definition of *language multiplication*, although it is not really a definition: it follows from the distributivity property.

The interpretation

A nonterminal can be seen as a variable of type language (like k, l are variables of type integer), a terminal is a constant language (like 3,5 are constant integers). Just like in the arithmetic example, we assume an assignment function that performs the mapping from symbols to their interpretation. This function is called L_G, as its value will be determined by the grammar G. Take, for example, the grammar rule $A \rightarrow xBy$. In the natural interpretation this rule means

$$xL_G(B)y \leftarrow L_G(A).$$

That is, the grammar rule is a constraint on L_G. In principle, L_G need only apply to nonterminals, but it is convenient to extend it, via

$$L_G(x) = x, \text{ for all } x \leftarrow V_T,$$

to all grammar symbols. Moreover, we further extend it to arbitrary strings of grammar symbols, via

$$\begin{aligned} L_G(\alpha\beta) &= L_G(\alpha)L_G(\beta), \\ L_G(\epsilon) &= \epsilon, \end{aligned} \qquad (9)$$

so that the following equalities hold true:

$$xL_G(B)y = L_G(x)L_G(B)L_G(y) = L_G(xBy).$$

Equation (9) states that L_G not only maps grammar-symbol strings into languages: it also maps an operation on its input objects (concatenation) to an operation on its output objects (language multiplication). In other words, the extended L_G is a homomorphism.

The interpretation of any grammar rule $A \rightarrow \alpha$ now reads

$$L_G(\alpha) \leftarrow L_G(A).$$

In other words, the language associated with A and the language associated with α are related: the latter is a sub-bunch of the former. Just like in the arithmetic example, a collection of such inequalities does not define the assignment function uniquely. There is one L_G, however, that assigns the smallest possible languages to grammar symbols. This smallest homomorphism is what the grammar is intended to define.

Inspired by the arithmetic analogue one may write X instead of $L_G(X)$ and insist that $A \rightarrow \alpha$ means $\alpha \leftarrow A$: α is a sub-bunch of A. Rules $A \rightarrow \alpha_1$, ...,$A \rightarrow \alpha_k$, for the same nonterminal, are often abbreviated to $A \rightarrow \alpha_1|...|\alpha_k$. This is very natural here; it means that $\alpha_1|...|\alpha_k$ is a subbunch of A.

A rule $A \rightarrow \alpha_1|...|\alpha_k$ involves a list of alternative strings. For completeness, and for later reference, let us give the formal semantics of more general rules $A \rightarrow a$, with a denoting arbitrary regular expressions. As above, this semantics is $L_G(a) \leftarrow L_G(A)$, where the application of L_G to

regular expressions is defined by (a, b are regular expressions)

$$\begin{aligned} L_G(ab) &= L_G(a)L_G(b), && \text{(concatenation)}\\ L_G(a|b) &= L_G(a) \mid L_G(b), && \text{(alternation)}\\ L_G(\langle a\rangle) &= \epsilon \mid L_G(a), && \text{(optionality)}\\ L_G(\{a\}) &= \epsilon \mid L_G(\{a\})L_G(a), && \text{(iteration)} \end{aligned} \tag{10}$$

where the brackets $\langle\ \rangle$ were used for optionality instead of the more usual [ ] to avoid confusion later on.

4 General recursive descent parsing

Given some input string ξ of terminal symbols, a grammar determines for each string of grammar symbols α whether or not ξ can be derived in any number of steps from α, i.e. whether $\alpha \xrightarrow{*} \xi$. Also, for each substring η of ξ it may be determined whether or not $\alpha \xrightarrow{*} \eta$. Let us define for each α a bunch-valued *recognition function* $[\alpha]$ from V_T^* to V_T^*, as follows:

$$[\alpha](\xi) = \alpha \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho. \tag{11}$$

Stated differently, this defines a function $[\cdot]$ that operates on two strings of grammar symbols, such that $[\cdot](\alpha, \xi) = [\alpha](\xi)$. Similar recognition functions, with lists instead of bunches, were introduced in (Wadler, 1985). Note that $\epsilon \leftarrow [S](\xi)$, equivalent to $S \xrightarrow{*} \xi$, means that ξ is a correct sentence.

In (11), the argument is split into two parts, the first of which is derivable from α. The second part is output by the function. It follows, for all α and β, that

$$\begin{aligned} [\alpha\beta](\xi) &= \alpha\beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho\\ &= \alpha \xrightarrow{*} \eta_1 \wedge \xi = \eta_1\rho_1 \wedge\\ &\quad\ \beta \xrightarrow{*} \eta_2 \wedge \rho_1 = \eta_2\rho \rhd \rho\\ &= \alpha \xrightarrow{*} \eta_1 \wedge \xi = \eta_1\rho_1 \rhd\\ &\quad\ (\beta \xrightarrow{*} \eta_2 \wedge \rho_1 = \eta_2\rho \rhd \rho)\\ &= \rho_1 \leftarrow [\alpha](\xi) \rhd [\beta](\rho_1)\\ &= [\beta]([\alpha](\xi)). \end{aligned}$$

Here (2) and (4) were used in the second and fourth transitions, respectively. Thus, $[\alpha\beta] = [\alpha][\beta]$, where $[\alpha][\beta]$ is the composition of functions $[\alpha]$ and $[\beta]$, defined by

$$(fg)(\xi) = g(f(\xi)). \tag{12}$$

In other words, $\alpha = X_1 \ldots X_k$ implies $[\alpha] = [X_1]\ldots[X_k]$ and $[\epsilon](\xi) = \xi$. In algebraic terms, the mapping $[\cdot]$ is a homomorphism from V^* to a function space of bunch-valued functions. As the functions $[\alpha]$ are compositions of functions $[X]$, an implementation for the latter implies an implementation of the former. Now,

$$\begin{aligned} [X](\xi) &= X \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho\\ &= ((X \leftarrow V_T \wedge X = \eta) \vee\\ &\quad\ (X \rightarrow \beta \wedge \beta \xrightarrow{*} \eta)) \wedge \xi = \eta\rho \rhd \rho\\ &= (X \leftarrow V_T \wedge \xi = X\rho \rhd \rho)|\\ &\quad\ (X \rightarrow \beta \wedge \beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho)\\ &= (X \leftarrow V_T \wedge \xi = X\rho \rhd \rho)|\\ &\quad\ (X \rightarrow \beta \rhd (\beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd \rho))\\ &= (X \leftarrow V_T \wedge \xi = X\rho \rhd \rho)|\\ &\quad\ (X \rightarrow \beta \rhd [\beta](\xi)). \end{aligned}$$

Here (3) was used to eliminate the disjunction $\vee$ and (2) to eliminate a conjunction $\wedge$. To summarize, we have, for terminals x and nonterminals A:

$$\begin{aligned} [x](\xi) &= \xi = x\rho \rhd \rho,\\ [A](\xi) &= A \rightarrow \alpha \rhd [\alpha](\xi),\\ [XY\beta](\xi) &= [Y\beta]([X](\xi)),\\ [\epsilon](\xi) &= \xi. \end{aligned} \tag{13}$$

Note the use of the distributivity property of bunches in the third line. If rules have regular expressions at their right-hand sides, all this is easily extended (compare with (10)):

$$\begin{aligned} [x](\xi) &= \xi = x\rho \rhd \rho,\\ [A](\xi) &= A \rightarrow a \rhd [a](\xi),\\ [ab](\xi) &= [b]([a](\xi)),\\ [a|b](\xi) &= [a](\xi) \mid [b](\xi),\\ [\langle a\rangle](\xi) &= \xi \mid [a](\xi),\\ [\{a\}](\xi) &= \xi \mid [\{a\}]([a](\xi)),\\ [\epsilon](\xi) &= \xi. \end{aligned} \tag{14}$$

The right-hand sides of lines three to six in (14) depend on a, b only via the functions $[a]$ and $[b]$. For this reason, these definitions are sometimes seen as applications of combinators, i.e., higher-order functions. With f, g denoting arbitrary bunch-valued functions from some domain (e.g., V_T^*) to itself, $\{f\}$, $[f]$, $f|g$ are other such functions, defined by

$$\begin{aligned} (f|g)(\xi) &= f(\xi) \mid g(\xi), && \text{(alternatives } f, g)\\ \{f\}(\xi) &= \xi \mid \{f\}(f(\xi)), && \text{(iterative } f)\\ \langle f\rangle(\xi) &= \xi \mid f(\xi). && \text{(optional } f) \end{aligned}$$

It follows that $[a|b] \equiv [a]|[b]$, $[\{a\}] \equiv \{[a]\}$, and $[\langle a \rangle] \equiv \langle [a] \rangle$. Finally, $[ab] \equiv [a][b]$, where $[a][b]$ is the functional composition of $[a]$ and $[b]$, defined in (12). In other words, the recognition function $[a]$ for regular expression a can be obtained by replacing every grammar symbol X that occurs in it by its function $[X]$ and interpreting all constructors in the regular expression (alternation, concatenation, iteration, optionality) as combinators of recognition functions. For a detailed exposition of combinator parsing, see (Hutton, 1992). (Norvell — Hehner, 1992) issued a warning that higher-order programming with bunch-valued functions may lead to paradoxes that were noted by (Meertens, 1986) in the case of nondeterministic functions. The above combinators do not suffer from problems of this kind.

5 Deterministic recursive descent parsing

The singleton property of bunches is notationally convenient if one applies a general parsing technique to grammars for which the technique happens to provide a deterministic recognizer. If the general technique is defined with set-valued recognition functions, in the deterministic case all these functions produce sets with at most one value. If a function produces the empty set, this means that an error has been detected. If one works with bunch-valued functions instead, in a deterministic recognizer these produce *null* if an error has occurred and definite values otherwise.

There is a standard method to make parsing algorithms more deterministic: the addition of look-ahead (Aho — Ullman, 1977). The application of look-ahead techniques to recursive descent parsing involves two functions, *first* and *follow*:

$$first(\alpha) = \quad x \leftarrow V_T \wedge \alpha \xrightarrow{*} x\beta \rhd x,$$

$$follow(X) = \quad A \rightarrow \alpha X \beta \rhd first(\beta) \mid \\ A \rightarrow \alpha X \beta \wedge \beta \xrightarrow{*} \epsilon \rhd follow(A).$$

Although *follow* not necessarily terminates if it is interpreted as an algorithm, it uniquely defines a smallest bunch $follow(X)$, for every X. It is convenient to add to each grammar the rule $S' \rightarrow S \perp$, where S' and $\perp$ are new symbols which appear only in this rule. S' is the new start symbol and $\perp$ is formally added to V_T. Of course, any correct input must now end with $\perp$. The above then implies that $\perp \leftarrow follow(S)$, and it is guaranteed that $follow(X) \neq null$ if $\exists_{\alpha\beta}(S \xrightarrow{*} \alpha X \beta)$. If X is one of the added symbols $S', \perp$ then $follow(X) = null$.

It is not difficult to verify that if for $A \neq S'$ function $[A]$ is redefined as

$$[A](\xi) = \quad A \rightarrow \alpha \wedge \xi = x\eta \wedge (x \leftarrow first(\alpha) \vee \\ (\alpha \xrightarrow{*} \epsilon \wedge x \leftarrow follow(A))) \rhd [\alpha](\xi),$$

the result of $[S'](\xi)$ is not affected. If for all $A \neq S'$ and every x at most one α exists that makes the guard true, the choice of grammar rule is always unique. This is the case for LL(1) grammars. For such grammars, the look-ahead technique makes each invocation $[A](\xi)$ produce either *null* if an error in the input string has been encountered, or a string of terminals that still have to be parsed; the general algorithm specializes to a fully natural deterministic recognizer.

6 Recursive ascent parsing

Bunch notation is equally useful for bottom-up parsing. To illustrate this, let us start from the following specification of an Earley-like parser $(\delta \leftarrow (V_T^* | V_N V_T^*))$:

$$[A \rightarrow \alpha \cdot \beta](\delta) = \\ \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\ \delta = X\zeta \wedge \beta \xrightarrow{*} X\eta \wedge \zeta = \eta\rho \rhd A\rho. \qquad (15)$$

If applied to a string ξ of terminal symbols, this specification reduces to

$$[A \rightarrow \alpha \cdot \beta](\xi) = \quad \beta \xrightarrow{*} \eta \wedge \xi = \eta\rho \rhd A\rho.$$

This means that, after adding a rule $S' \rightarrow S$ to the original grammar, it follows that

$$S' \leftarrow [S' \rightarrow \cdot S](\xi)$$

if and only if ξ is a correct sentence. The intuition behind this is that a function invocation $[A \rightarrow \alpha \cdot \beta](\delta)$ investigates which prefixes of δ can be rewritten to β, in a bottom-up way (using grammar rules from right to left). If a non-empty prefix can be found, this corresponds to a part of the input sentence, which is a string rewritable to the first symbol of δ (i.e. X) followed by the remainder of the prefix (which are terminals). If

$\beta \xrightarrow{*} \epsilon$, the prefix may be empty. Assuming that the function is invoked only if a directly preceding part of the input sentence was rewritable to α, it is deduced that this preceding part, followed by the part that corresponds to the found prefix of δ, can be rewritten into A. The function thus returns A, followed by the part of the input sentence that has not yet been parsed. If more than one prefix can be found, the function delivers a bunch.

We strive for an implementation of (15) of the recursive ascent type. To this end, we note that $\beta \xrightarrow{*} X\eta$ means that either X is introduced by a grammar rule $B \rightarrow \mu X\nu$, with $\mu \xrightarrow{*} \epsilon$, or X is already in β: $\beta = \mu X\nu$, with $\mu \xrightarrow{*} \epsilon$:

$$
\begin{aligned}
&[A \rightarrow \alpha \cdot \beta](\delta) = \\
&\quad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\
&\quad \delta = X\zeta \wedge \beta = \mu X\nu \wedge \mu \xrightarrow{*} \epsilon \wedge \nu \xrightarrow{*} \eta \wedge \\
&\qquad \zeta = \eta\rho \rhd A\rho \mid \\
&\quad \delta = X\zeta \wedge \beta \xrightarrow{*} B\eta_1 \wedge B \rightarrow \mu X\nu \wedge \\
&\qquad \mu \xrightarrow{*} \epsilon \wedge \nu \xrightarrow{*} \eta_2 \wedge \zeta = \eta_2\eta_1\rho \rhd A\rho.
\end{aligned}
$$

After a few elementary rewriting steps using (15), one finally obtains

$$
\begin{aligned}
&[A \rightarrow \alpha \cdot \beta](\delta) = \\
&\quad \delta \leftarrow V_T^* \wedge \beta \xrightarrow{*} \epsilon \rhd A\delta \mid \\
&\quad \delta = X\zeta \wedge \beta = \mu X\nu \wedge \mu \xrightarrow{*} \epsilon \rhd \\
&\qquad [A \rightarrow \alpha\mu X \cdot \nu](\zeta) \mid \\
&\quad \delta = X\zeta \wedge \beta \xrightarrow{*} B\gamma \wedge B \rightarrow \mu X\nu \wedge \mu \xrightarrow{*} \epsilon \\
&\qquad \rhd [A \rightarrow \alpha \cdot \beta]([B \rightarrow \mu X \cdot \nu](\zeta)).
\end{aligned}
\tag{16}
$$

The conciseness of the last line is due to the distributivity property of bunches. In deriving (16) a critical need is that not $B \leftarrow V_T$, in other words, that terminals and nonterminals are disjoint.

Note that if a function $[A \rightarrow \alpha \cdot \beta]$ is invoked by another function, then its argument δ is in V_T^*. It may recursively call itself with rewritten versions of this δ, i.e., with prefixes of δ replaced by some nonterminal B, until this B appears in β in such a way that the symbols before B (in β) may derive the empty string.

The recognizer terminates for all non-cyclic grammars. Note that the conditions

$$
\begin{aligned}
&\beta \xrightarrow{*} \epsilon \\
&\beta = \mu X\nu \wedge \mu \xrightarrow{*} \epsilon \\
&\exists_\gamma (\beta \xrightarrow{*} B\gamma) \wedge B \rightarrow \mu X\nu \wedge \mu \xrightarrow{*} \epsilon
\end{aligned}
$$

are independent of the input string, and for every β, X the values of μ, ν, B that make them true can be computed before parsing. To get an efficient implementation such pre-computation is to be compounded with function memoization (Leermakers, 1992; 1993).

In the case of a grammar without ϵ-rules, (16) becomes even simpler:

$$
\begin{aligned}
&[A \rightarrow \alpha \cdot \beta](\delta) = \\
&\quad \delta \leftarrow V_T^* \wedge \beta = \epsilon \rhd A\delta \mid \\
&\quad \delta = X\zeta \wedge \beta = X\nu \rhd [A \rightarrow \alpha X \cdot \nu](\zeta) \mid \\
&\quad \delta = X\zeta \wedge \beta \xrightarrow{*} B\gamma \wedge B \rightarrow X\nu \rhd \\
&\qquad [A \rightarrow \alpha \cdot \beta]([B \rightarrow X \cdot \nu](\zeta)).
\end{aligned}
$$

As far as I know, the recognizer of this section is a new variant of Earley-like parsing. In (Leermakers, 1992) a closely related algorithm was given, with two functions per dotted rule, instead of one. The functional parsing algorithm given in (Matsumoto et al., 1983) is also quite similar to ours, even though it does not involve dotted rules. For a discussion of the relation of the above algorithm with the standard Earley parser, see (Leermakers, 1993). An analogous LR parser, with one function for each state (and, of course, without a parse stack), is also constructed in (Leermakers, 1993).

7 Conclusions

This paper should serve two purposes. Firstly, it should show the beauty of functional parsing theory. Secondly, the paper is meant to establish, by way of illustrative examples, that the bunch concept is a mathematical notion as respectable as sets and lists. The reader is invited to translate any of the sections into set notation and observe the notational burden that he/she has to add.

One could argue that almost the same conciseness can be obtained using normal sets and an extra ('map') operator to distribute functions over sets. However, one should keep in mind that the bunch notion is more primitive than its set relative: a bunch is an aggregation, a set is an encapsulated aggregation (Hehner, 1993). It is the encapsulation aspect of sets that leads to conceptual problems, to students (a set that contains nothing is not nothing) as well as to scientists (the set that contains everything, including itself, leads to a paradox). Being essentially simpler,

bunches are not troubled by such intricacies. In practice, it is fine to implement bunches with sets, as long as one keeps in mind the difference between a notion and its implementation. After all, the possibility of implementing sets in terms of lists does not mean that sets can be dispensed with. One distinguishing aspect of bunch-valued functions, which goes beyond notational issues, is that normal functions are embedded in them.

The conciseness of bunch notation is not its only virtue. Functions defined with bunch notation look more 'algorithmic' than their translation into set notation, which is not unimportant if one wants to define an algorithm, if only for pedagogical reasons.

The notion of bunches has been introduced in (Hehner, 1984). Sets with nondeterministic interpretation, like bunches, were also proposed in (Hughes — O'Donnell, 1990). In (Wadler, 1992) a kind of bunch-valued lambda-calculus is discussed. Bunch-valued functions also appear in (Meertens, 1986; Bauer et al., 1987; Norvell — Hehner, 1992), as nondeterministic specifications of programs.

I refer to (Hehner, 1993) for further elaborations on the bunch theme, and many other applications. This work also proposes to make a distinction between strings and sequences, which also exists between bunches and sets: strings have the singleton property, but sequences do not. As is apparent from the notation for elements of V^*, it is natural to make no distinction between grammar symbols and elements of V^* that have length one. Thus, elements of V^* are strings, not sequences. In (Leermakers, 1993) bunch notation is adopted as a tool for the formulation of parsing theory, in the spirit of this paper. In this book, bunches are also used in the theory of attribute grammars. In conventional attribute grammars, each attribute has an associated function that computes its value in terms of the values of other attributes. It is very natural to take such an attribute function to be bunch-valued. If the function produces *null*, this means that the computation of its attribute fails. If it produces a bunch with more than one element, attribute computation is ambiguous. Bunch-valued attribute functions are particularly apt for natural-language parsing, since both failure and ambiguity of attribute computation are natural phenomena in this application of attribute grammars.

Acknowledgement

I thank Theo Norvell for his useful comments on the first draft of this paper, and Lex Augusteijn, Paul Jansen, Frans Kruseman Aretz and Mark-Jan Nederhoffor their constructive remarks about the second draft. Triggered by (Norvell — Hehner, 1992), it was Lex Augusteijn who inspired me to use bunches for the kind of parsing algorithms we are both engaged in.

References

Aho A.V. — J.D. Ullman (1977) *Principles of Compiler Design.* Reading, MA: Addison-Wesley.

Bauer F.L. — H. Ehler — A. Horsch — B. Möller — H. Partsch — O. Puakner — P. Pepper (1987) *The Munich Project CIP: Volume II: The Program Transformation System CIP-S.* Lecture Notes in Computer Science 292. Berlin: Springer-Verlag.

Dijkstra E.W. (1976) *A Discipline of Programming.* London: Prentice Hall.

Hehner E.C.R. (1984) *The Logic of Programming.* London: Prentice-Hall.

Hehner E.C.R. (1993) *a Practical Theory of Programming.* Berlin: Springer-Verlag.

Hughes J. — J. O'Donnell (1990), "Expressing and reasoning about non-deterministic functional programs". In: K. Davis and J. Hughes (Eds), *Functional Programming.* Berlin: Springer-Verlag.

Hutton G. (1992) "Higher-order functions for parsing". In: *Journal of Functional Programming* 2(3), 323–343.

Leermakers R. (1992) "A recursive ascent Earley parser". In: *Information Processing Letters* 41, 87–91.

Leermakers R. (1993) *The Functional Treatment of Parsing.* Amsterdam: Kluwer Academic Publishers.

Matsumoto Y. — H. Tanaka — H. Hirakawa — H. Miyoshi — H. Yasukawa (1983) "BUP: a bottom-up parser embedded in Prolog" *New Generation Computing* 1(2).

Norvell T.S. — E.C.R. Hehner (1992) "Logical Specifications for Functional Programs". In: *Proceedings of the Second International Conference on the Mathematics of Program Construction.* Oxford: Oxford University Press.

Sondergard — Sesoft (1990) "Referential Transparency, Definiteness and Unfoldability". *Acta Informatica* 27, 505–517.

Wadler P. (1985) "How to replace failure by a list of successes". In: *Conference on Functional Programming Languages and Computer Architecture* (Nancy, France); LNCS 201. Berlin: Springer-Verlag.

Wadler P. (1992) "The essence of functional programming", In: *19th Annual Symposium on Principles of Programming Languages* Santa Fe.

Chart Parsing of Attributed Structure-Sharing Flowgraphs with Tie-Point Relationships

Rudi Lutz

School of Cognitive and Computing Sciences, University of Sussex
Falmer, Brighton BN1 9QH, England
email: rudil@cogs.susx.ac.uk

Abstract

Many applications make use of diagrams to represent complex objects. In such applications it is often necessary to recognise how some diagram has been pieced together from other diagrams. Examples are electrical circuit analysis, and program understanding in the plan calculus (Rich, 1981). In these applications the recognition process can be formalised as flowgraph parsing, where a flowgraph is a special case of a plex (Feder 1971). Nodes in a flowgraph are connected to each other via intermediate points known as tie-points. Lutz (1986, 1989) generalised chart parsing of context-free string languages (Thompson — Ritchie, 1984) to context-free flowgraph languages, enabling bottom-up and top-down recognition of flowgraphs. However, there are various features of the plan calculus that complicate this - in particular attributes, structure sharing, and relationships between tie-points. This paper will present a chart parsing algorithm for analysing graphs with all these features, suitable for both program understanding and digital circuit analysis. For a *fixed* grammar, this algorithm runs in time polynomial in the number of tie-points in the input graph.

1 Introduction and Motivation

Many applications make use of diagrams to represent complex objects, and we often need to recognise how some diagram has been constructed. Examples are electrical circuit analysis, and program understanding in the plan calculus (Rich, 1981), in which programs are represented by data- and control- flow graphs, and stereotypical programming techniques and algorithms (plans) are represented similarly. Understanding how a program has been built up then amounts to treating plans as forming a grammar, and the understanding process as parsing. Ignoring control flow connections enables us to formalise this as flowgraph parsing. Nodes in a flowgraph consist of labelled boxes with distinguished input and output attaching points (ports), and input ports are connected to output ports via intermediate points known as tie-points, with the restriction that a port is only ever connected to a single tie-point, although fan-out and fan-in is allowed at tie-points. Lutz (1986, 1989) generalised chart parsing of context-free string languages (Thompson — Ritchie, 1984) to context-free flowgraph languages, enabling bottom-up and top-down recognition of flowgraphs. However, there are features of the plan calculus that complicate this:

1. Attributes — control flows in the plan calculus are treated as attributes of the grammar which are propagated during parsing.
2. Data Plans and Overlays — Some plans in the plan calculus allow the introduction of new tie-points not in the input graph. These tie-points either represent aggregate data structures corresponding to collections of other tie-points, or represent a more abstract view of some tie-point (e.g. viewing a list as implementing a set), and act as inputs or outputs of "higher-level" operations. Dealing with this involves using a

second chart storing information about data objects.

3. Structure Sharing — when one component feeds one or more of its outputs to more than one other component (fan-out). In this situation the source component can be viewed as playing more than one role in the structure, and could have been duplicated so that separate copies of the component were responsible for each of these roles. This leads to no change in functionality, although there may be a loss in efficiency as measured by the number of components (digital circuits), or computational effort and code size (plan calculus).

This paper will present a parsing algorithm for analysing graphs with these features, noting that we *permit* structure sharing, but do *not enforce* it. For a *fixed* grammar, this algorithm runs in time polynomial in the number of tie-points in the input graph. We will begin by discussing simple flowgraphs, and then progressively deal with the above features.

2 Notation and Definitions

Flowgraphs and *flow grammars* are special cases of plex languages and plex grammars (Feder, 1971). A *plex* consists of labelled nodes having an arbitrary number, n, of distinct *attaching points*, used to join nodes together. Such a node is called an n-attaching point entity (NAPE). Attaching points of NAPEs do not connect directly, but via intermediate points known as tie-points. A single tie-point may connect two or more attaching points. If the direction of the connections is important then the plex is said to be directed. Many types of graph structure (e.g. webs (Pfaltz — Rosenfeld, 1969; Rosenfeld — Milgram, 1972), directed graphs, and strings) are special cases of directed plexes. We will consider the special case of directed plexes in which each NAPE's attaching points (from now on called *ports*) are subdivided into two mutually exclusive groups, known as *input ports* (restricted to only have incoming connections) and *output ports* (restricted to only have outgoing connections). We further restrict ourselves to the case in which each port of a NAPE is only connected to a single tie-point. This type of plex will be called a *flowgraph* and is a generalisation of Brotsky's (1984) use of the term. See Figure 1 (top) for a simple example.

A production in a string grammar specifies how one string may be replaced by another. However, with flowgraph grammars we encounter a difficulty (due to their 2-dimensional nature) not apparent in the string case. In the string case a production

$$A ==> aXYb$$

applied to a string

$$\ldots dAe \ldots (say)$$

results in the string

$$\ldots daXYbe \ldots$$

and the question of how the replacement string is embedded in the host string never arises because there is a single obvious choice i.e. whatever is to the left of A in the original string is to the left of the replacing string, and similarly on the right. With flowgraphs we no longer have this simple ordering on the NAPEs and embedding becomes much more complicated. Most of the discussion of this is in the web and graph grammar literature (e.g. (Pfaltz — Rosenfeld, 1969; Rosenfeld — Milgram, 1972)), but most of it also applies to flowgraphs. Our approach is to specify with each production which tie-points on the left correspond to which tie-points on the right and then connect everything connecting to one of these left hand tie-points (from the surrounding subgraph) to its corresponding right-hand tie-point.

We define a flowgraph grammar to be a 4-tuple (N,T,P,S) where:

N is a finite non-empty set of NAPEs known as nonterminals.

T is a finite non-empty set of NAPEs known as terminals.

P is a finite set of productions.

S is a special member of N known as the initial, or start, NAPE

where the intersection of N and T is empty.

If we arbitrarily order the input and output ports of a NAPE then each NAPE in a flowgraph can be represented as a triple

$$(NAPE - label, inputlist, outputlist)$$

where NAPE-label is the label on the NAPE, and input list is a list in which the i^{th} entry is the tie-point to which the i^{th} input port is connected. Similarly the output list specifies to which tie-point each of the output ports is connected. Using this convention a flowgraph G can be represented as a set G^c (the component set) of such triples.

With the above conventions the productions in a flowgraph grammar have the general form

$$AL_iL_o ==> CR_iR_o$$

where

- A is known as the left-side structure, represented as a component set
- C is known as the right-side structure, represented as a component set
- L_i is the left-side input tie-point list
- R_i is the right-side input tie-point list
- L_o is the left-side output tie-point list, and
- R_o is the right-side output tie-point list.

L_i and R_i must be of the same length, as must L_o and R_o , and specify how an instance of the right-side structure is to be embedded into a structure W containing an instance of the left-side structure which is being rewritten according to the production. We define the *arity* of the left side of the rule to be the ordered pair $(|L_i|, |L_o|)$ and the arity of the right side of the rule to be the ordered pair $(|R_i|, |R_o|)$. So this requirement simply states that the left- and right-side arities must be the same. The rewriting and embedding is done as follows:

The instance of the left-side structure is removed from W and replaced by an instance of the right-side structure. Now, for each tie-point X in L_i any previous connections from NAPEs in W to X are replaced by connections from the same attaching points of the same NAPEs to the corresponding tie-point in R_i . The same is done for tie-points in L_o and R_o. One can eliminate the need for explicit storing of R_i and R_o by using the same variable names on the left and right hand sides of the production to denote corresponding tie-points.

Just as in the string case, by considering restrictions on X and Y in a production of the form:

$$X ==> Y$$

one can arrive at the notions of context-sensitive, context-free, and regular grammars (Ehrig, 1979). In particular, restricting the productions to have a single NAPE in their left-side structure gives us a context-free flowgraph grammar, and we will restrict ourselves to these from now on. In this case we no longer need to store L_i and L_o since the input and output lists of the single triple on the left of the production already specify this information. See Figure 1 for an example of the notation and of rewriting process.

3 Chart Parsing of Context-free Flowgraphs

In a chart parser, assertions about what has been found by the parser are kept in a "database" known as the *chart*. Such assertions will be called *patches*, and are of two kinds — *complete* patches and *partial* patches. A complete patch asserts that a complete grammatical entity (corresponding to some terminal or non-terminal symbol of the grammar) has been found. Partial patches are assertions that part of some grammatical entity has been found, and about what needs to be found to complete it. One can think of a patch as being a closed loop drawn round some subgraph of the flowgraph, indicating that this subgraph corresponds to all or part of some grammatical entity. Regarding the right-side structures of rules as uninstantiated templates, then complete patches with non-terminal labels correspond to the occurrence of an instantiation of the right-side structure of some rule, thus forming an occurrence of the left-side structure of the rule. Partial patches correspond to partially instantiated instances of the right-side structure of some rule, and thus to partially recognised instances of the left-side structure. Each patch A contains the following information:

1. *label(A)* — the name (one of the terminal or non-terminal symbols) of the grammatical entity corresponding to the patch.

2. *inputs(A)* — a set of input tie-points for the patch.

3. *outputs(A)* — a set of output tie-points for the patch.

4. *components(A)* — a list of the patches involved in making up this patch.

5. *needed(A)* — what else needs to be found to complete the patch. For complete patches this will be empty, and for partial patches this will be a flowgraph structure, represented as a list of triples.

For a partial patch, the input and output tie-points (i.e. those by which the patch connects to the surrounding flowgraph) are each subdivided into two categories — the set of *active* tie-points where the patch still needs other components to attach to these tie-points, and the set of *inactive* tie-points which are those which would be inputs or outputs of the patch were it complete. A NAPE needed by a partial patch will be called *immediately needed* if any of its tie-points are active. The components entry of a patch lists (instantiated versions of) those NAPEs in the right-side structure of the rule which have been completely instantiated, and the needed entry lists uninstantiated (as yet) parts of the rule. Note that some of the *tie-points* in the *needed entry* may be instantiated. These are where the needed NAPEs connect to the ones already found. We will say that a partial patch A is *extendible* by a complete patch B (or that B *can extend* A) in the case where A immediately needs a patch of the same type as B and the instantiated tie-points in this needed patch do not conflict with any instantiations actually occurring in B.

The essence of the chart parsing strategy can then be stated as follows:

Every time a complete patch is added to the chart a search is made for any partial patches immediately needing a patch like the one just added. For each of these partial patches a *new* patch is made extending it by the complete one, and this new patch is then added to an agenda of patches to be processed at some appropriate time. Similarly, every time a partial patch is added to the chart a search is made for complete patches which can extend the partial patch just added, and if any are found new patches are made extending the partial one, and these are added to the agenda to be processed when appropriate. Note that patches are only ever added to the chart. They are never removed, thus avoiding duplication of previous effort.

The basic operation of the algorithm is joining a complete patch to a partial patch to make a new enlarged patch. Figure 2 shows a partial patch being joined to a complete patch to make a new patch (the enclosing box). The resulting patch has the same items in its *components entry* as the original partial patch plus the complete patch. Its *needed entry* is equal to that of the original partial patch minus the needed patch corresponding to the complete patch. Note that the matching of a needed patch to an actual complete patch may introduce further instantiations of tie-points in the *needed entry* of the new patch. On connecting the two patches all the inactive tie-points of the partial patch remain inactive. Some of its active tie-points will correspond to tie-points of the complete patch (this is where the two patches actually join). Other active tie-points remain active in the new patch since it is still looking for other patches to attach to them. Of the complete patch's (input and output) tie-points some have already been mentioned i.e. those connecting directly to the partial patch. Others will become new inactive tie-points of the resulting patch since it will not be looking for anything to attach to them. However other (input and output) tie-points of the complete patch may now become active (viewed as belonging to the new patch) since it may now expect other patches to attach to them in order to complete itself. Provided all these distinctions are kept clear there is no great difficulty in implementing the joining operation.

With this joining operation a limited type of structure sharing happens automatically. This is illustrated in Figure 3. If we wish to prevent this, then, when trying to extend a partial patch P by a complete patch C, the parser must check (recursively!) that none of the components of P have any sub-components in common with C, thus preventing structure sharing at any level. This check will be referred to as the *no-sharing check.*

The initialisation of the agenda will now be described. Initially a complete patch is added to the agenda for each of the terminal NAPEs in the original graph. If the algorithm is to run top-down then an additional step is needed in which partial patches with empty *components entries*

are made for every rule in the grammar whose left-side structure is labelled by the start symbol of the grammar. Each such rule leads to several such empty patches, one for each permutation of the input tie- points of the input graph. The *inactive-inputs* and *active-outputs* entries for these patches are the permuted inputs. The *needed entry* is the right-side structure of the rule with appropriate instantiations of the tie points occurring in it. These patches are also added to the agenda. The complete algorithm is shown below:

```
initialise chart and agenda;
until the agenda is empty do
    pick a patch A from the agenda;
    unless A is already in the chart then
        add A to the chart;
        if A is complete then
        for each partial patch B in chart extendible by A do
                make a new patch extending B with A and put on agenda;
            endfor;
            if bottom-up then
                for each rule R in P such that rhs(R) has an input NAPE  labelled by
                                        label(A) do
                  for each such NAPE X in R do
                    make new empty patch B with label(B)=lhs(R) and
                        needed(B)=rhs(R) with instantiations dependent on match between
                                              X and A and
                        inputs(B)=inputs(A) and
                        active-outputs(B)=inputs(A);
                     add B to agenda;
                  endfor;
                endfor;
            endif;
        else
            for each complete patch B in chart which can extend A do
                make a new patch extending A with B and put on agenda;
            endfor;
            if top-down then
                for each object C immediately needed by A do
                    for each rule R in P with lhs(R)=label(C) do
                          make new empty patch B with label(B)=label(C) and
                              needed(B)=rhs(R) with instantiations dependent on match
                                        between C and lhs(R) and
                              inputs(B)=inputs(C) and
                              active-outputs(B)=inputs(C);
                          add B to agenda;
                    endfor
                endfor
            endif
        endif
     endunless
enduntil;
```

$$A_{N+1} \begin{cases} = & A_N - 1 \quad \textit{if patch is already present in the chart} \\ \leq & A_N - 1 + AQT^{K+M-1} + QR \quad \textit{if patch partial and not in chart} \\ \leq & A_N - 1 + (K+M) \cdot R2^Q T^{K+M+A-1} \quad \textit{if patch complete and not in chart} \end{cases}$$

$$C_{N+1} \begin{cases} = & C_N \quad \textit{if patch chosen is already present in the chart} \\ = & C_N \quad \textit{if patch chosen is partial and not in the chart} \\ = & C_N + 1 \quad \textit{if patch chosen is complete and not in the chart} \end{cases}$$

$$P_{N+1} \begin{cases} = & P_N \quad \textit{if patch chosen is already present in the chart} \\ = & P_N + 1 \quad \textit{if patch chosen is partial and not in the chart} \\ = & P_N \quad \textit{if patch chosen is complete and not in the chart} \end{cases}$$

Figure 14: The top-down case, the $N+1^{th}$ iteration

On termination the parsing is successful if the chart contains a complete patch for S whose *inputs* and *outputs* entries are the input and output tie-points of the input graph.

How can we organise the chart for efficient searching? The chart is divided into two parts, one for complete patches, and one for partial. The part for complete patches is organised as two arrays, one for indexing each patch by its inputs, and one for indexing by its outputs. So each complete patch is entered several times into the chart, once for each of its inputs and outputs. For further efficiency each of the elements in these arrays is a hash table and the patches are actually entered into these hashed by their label. So the entries in the hash table are actually lists of patches with the same label which share a given input or output tie-point. This enables efficient retrieval of all patches with a particular label at a particular tie-point. The treatment of partial patches is slightly more complicated. For each of their immediately needed NAPEs partial patches are entered into their part of the chart indexed by the active inputs and outputs of the needed NAPE, and hashed by the labels of each of these NAPEs. This structure for the chart enables a complete patch to easily find partial patches immediately needing it, and enables partial patches to easily find complete patches that they immediately need.

A similar technique can be used to store the grammar rules in order to enable efficient retrieval of appropriate rules.

4 Complexity Analysis

4.1 A Polynomial Bound

In this section a relatively informal argument will be given to show that, for a *fixed grammar*, the algorithm runs in time polynomial in the number of tie-points T of the input graph (if the grammar is allowed to vary and is therefore regarded as part of the input to the parsing problem, then Wills (1992) has shown that the problem becomes NP-complete). We will not give a tight upper bound on the running time, but simply show that it is polynomial. Let:

G =number of NAPEs in the graph

T =number of tie-points in the graph

K =maximum number of inputs to a NAPE

M =maximum number of outputs from a NAPE

L =number of possible labels

R =number of rules in the grammar

Q =maximum number of NAPEs in the right-side structure of a rule

A =maximum possible number of active tie-points in a partial patch.

Note that K, M, L, R, Q, and A all depend on the grammar, and are independent of the input graph.

$$A_{N+1} \begin{cases} = & A_N - 1 \quad \textit{if patch chosen is already present in the chart} \\ \leq & A_N - 1 + A \cdot Q \cdot T^{K+M-1} \quad \textit{if patch partial and not in chart} \\ \leq & A_N - 1 + (K+M) \cdot R2^Q T^{K+M+A-1} + QR \quad \textit{if patch complete and not in chart} \end{cases}$$

$$C_{N+1} \begin{cases} = & C_N \quad \textit{if patch chosen is already present in the chart} \\ = & C_N \quad \textit{if patch chosen is partial and not in the chart} \\ = & C_N + 1 \quad \textit{if patch chosen is complete and not in the chart} \end{cases}$$

$$P_{N+1} \begin{cases} = & P_N \quad \textit{if patch chosen is already present in the chart} \\ = & P_N + 1 \quad \textit{if patch chosen is partial and not in the chart} \\ = & P_N \quad \textit{if patch chosen is complete and not in the chart} \end{cases}$$

Figure 15: The bottom-up case, the $N+1^{th}$ iteration

For the purposes of adding new patches to the chart, patches are only distinguished according to some of the information contained in them, rather than strict equality being necessary. Complete and partial patches will be dealt with separately.

Complete patches are distinguished which differ in at least one of their input tie-points, their output tie-points, or their label. The maximum number of inputs and outputs in a patch is determined by the grammar, as is the number of possible labels. So the number of possible complete patches in the chart is bounded above by the product of L and the number of possible ways of selecting at most K out of T tie-points, and the number of possible ways of choosing at most M out of T tie-points. This gives us $O(L \cdot T^{K+M})$ complete patches altogether. A similar argument shows that at a given set of K (input) tie-points, there are *at most* $O(T^M)$ complete patches with a given label.

Partial patches are distinguished which differ in at least one of their inactive input tie-points, their inactive output tie-points, their label, or in what they need in order to complete themselves (their *needed* entry). Now, a partial patch represents the partially recognised right side structure of a rule. The rule used determines the label, and there are at most 2^Q subsets of the (at most) Q NAPEs in the rule that could still be needed. Each such subset determines a set of (at most) A active tie-points for the patch. So there can be at most

$$O(R \cdot 2^Q \cdot T^A \cdot T^K \cdot T^M) = O(R \cdot 2^Q \cdot T^{A+K+M})$$

partial patches altogether. In fact there will be very much less than this, as this includes complete patches with nothing needed, and (more importantly) ignores completely any additional constraints implied by the connectivity of the graph.

Since the basic operation of the chart parsing algorithm involves extending partial patches by complete ones, we need to know, for a given partial patch, the largest number of complete patches that could possibly extend it. A partial patch can be extended at any of its (at most) A active tie-points, and any complete patch which could extend it must join at least one of these tie-points, and must share a label with at least one of the NAPEs immediately required by the partial patch. So there are *at most* $O(A \cdot Q \cdot T^{K+M-1})$ such complete patches. Similarly, given a complete patch, there are at most $O((K+M) \cdot R \cdot 2^Q \cdot T^{K+M+A-1})$ possible matching partial patches.

We can use these upper bounds to demonstrate that the algorithm terminates, and does so in polynomial time. Let N denote the number of iterations of the main loop of the algorithm while it is running, and let:

C_N =number of complete patches in the chart after iteration N

P_N =number of partial patches in the chart after iteration N

A_N =length of agenda after iteration N

Then in the top-down case we have:[1]

$$\begin{aligned} A_0 &= G + R_S \cdot P_G \\ C_0 &= 0 \\ P_0 &= 0 \end{aligned}$$

The equation for the $(N+1)^{th}$ iteration is in Figure 14. In the bottom-up case we have

$$\begin{aligned} A_0 &= G \\ C_0 &= 0 \\ P_0 &= 0 \end{aligned}$$

and for the $(N+1)^{th}$ iteration see Figure 15.

So, in both the bottom-up and top-down cases, C_N and P_N are monotonic functions of N. As discussed earlier both are bounded above. Therefore after some number of iterations they must both have reached their maximum value (normally *much less* than the crude estimates above). Once this happens all patches on the agenda must be already present in the chart and A_N decreases by one on each subsequent iteration until it reaches 0 (an empty agenda), and the algorithm terminates. Now on each iteration it can be seen that either:

1. both C_N and P_N remain constant (in which case A_N decreases), or
2. P_N increases by 1, and items are possibly added to the agenda, or
3. C_N increases by 1, and items are possibly added to the agenda.

From the above, at most $O(L \cdot T^{K+M})$ iterations involve adding a complete patch to the chart and add some items to the agenda, and at most $O(R \cdot 2^Q \cdot T^{A+K+M})$ iterations involve adding a partial patch to the chart and add some items to the agenda. All other iterations simply remove items from the agenda. So how many items get added to the agenda?

This is given by:

(no. of items in initial agenda)
+(no. added for complete patches)
+(no. added for partial patches)

In the top down case this bounded by

$$A0 + O(L \cdot T^{K+M}) \cdot O((K+M) \cdot R \cdot 2^Q \cdot T^{K+M+A-1}) + O(R \cdot 2^Q \cdot T^{A+K+M}) \cdot (O(A \cdot Q \cdot T^{K+M-1})) + Q \cdot R$$

and in the bottom-up case this is bounded by

$$A0 + O(L \cdot T^{K+M}) \cdot (O((K+M) \cdot R \cdot 2^Q \cdot T^{K+M+A-1}) + Q \cdot R) + O(R \cdot 2^Q \cdot T^{A+K+M}) \cdot O(A \cdot Q \cdot T^{K+M-1})$$

which are both clearly polynomial.

So, in both cases the number of items added to the agenda, which is the same as the number of iterations performed, is polynomially bounded. How much work is done on each of these iterations? The cost of seeing if a patch is already in the chart can be done in polynomial time. This is because (even with no clever indexing) there are at most a polynomial number of patches in the chart that need to be checked. If the no-sharing check is included then the cost of checking if one patch is extendible by another can be done in time at worst O(G) since both the partial and complete patches are each ultimately made up of at most G NAPEs (at lowest level), and checking for intersection of these two sets can be done in linear time. If the no-sharing check is omitted then the cost of checking if one patch is extendible by another can be done in constant time (since it depends on checking that the instantiated tie-points of the patches are compatible with each other, and the number of tie-points involved depends on the grammar), as can the cost of making a new patch. All the costs involved in checking rules etc. are purely a function of the grammar. So the total cost of the algorithm is easily seen to have an upper bound which is a polynomial function of T.

4.2 Finding All Parses

Athough the algorithm performs flowgraph *recognition* in polynomial time, it does not find *all parses* in polynomial time. This is because for some flowgraphs and some grammars there may well be an exponential number of parses (this is even true of Earley's algorithm operating on strings!). The algorithm will however find a parse if one exists. If an application requires all possible parses, then the algorithm can be modified to

[1] R_S is the number of rules for S (the start symbol); P_G is the number of permutations of inputs of graph.

store any *complete* patch which is equal to one already in the chart in terms of its inputs, outputs, and label, but *not* equal in terms of its *components*, in an auxiliary data structure. At the end of the parsing there is then enough information around in the chart and the auxiliary data structure to easily compute additional parses, by simply adding all the patches in the auxiliary structure to the agenda, and letting the parsing continue with the test for equality of patches now being strict equality (i.e. all the components must be equal as well) rather than just the partial equality used earlier.

5 Dealing With Attributes

As stated earlier graphs and rules in the plan calculus also have a second type of connection between NAPEs - control flow arcs. These are handled as attributes of the graphs, where the attribute for a non-terminal NAPE is calculated from the attributes of its components. Details of this method of handling the control flows can be found in Wills (1986, 1990, 1992), and a generalisation can be found in Lutz (1992). For our purposes we will assume that each NAPE in the original graph is annotated with initial values for the attributes, and we will also assume that each rule has annotations describing how each attribute for the left-hand side of the rule is computed from the attributes of the NAPEs on its right-hand side. These annotations have the general form:

$$A_{lhs} = f_{Rule}(A_1, \ldots, A_k)$$

where A_{lhs} represents an attribute of the left hand side of the rule, f_{Rule} represents the rule specific computation, and $A_1, \ldots, A_k$ represent the attributes of the NAPEs on the right.

Computing the attributes is straightforward. Each patch is given an extra field for each of its attributes. When a *complete* patch (corresponding to some rule of the grammar) whose components have attributes $A_1, \ldots, A_k$, is added to the chart, $f_{Rule}(A1, \ldots, Ak)$ is computed, and stored in the appropriate field for the attribute in the patch. The initial patches receive their attribute values from the original graph.

6 Dealing With Tie-Point Relationships

In order to capture implementation decisions, and data abstractions, the plan calculus contains what Rich (1981) calls data plans and data overlays. So far as the grammatical formalism is concerned, these can be viewed as allowing rules to express named functional relationships that hold between tie-points. To handle these our grammatical formalism is extended to allow annotations (following the keyword *where*) of the form:

$$t_i = F(t_{j_1}, \ldots, t_{j_k}) \quad for\ k \geq 1$$

where t_i represents either any of the tie-points occurring in the NAPEs of the rule, or an additional new tie-point, $t_{j_1}, \ldots, t_{j_k}$ represent any of the tie-points occurring in the NAPEs of the rule or any new tie-points mentioned on the left of other relationships in the rule (this must be non-recursive!), and F is the name of the functional relationship involved. The set of these will be referred to as the *tie-point relationships* of the rule.

We will only discuss the changes to deal with tie-point relationships for the parser running in bottom-up mode. Dealing with them in top-down mode is rather complicated and will not be described further in this paper.

Conside the rules *bump+update* and *bump+update->push* (Figures 4 and 5), which cause problems for the algorithm. Flowgraph grammar rules as described earlier have the same arities for their left- and right-hand sides, and this is true for *bump+update*. However, the left-hand side of *bump+update->push* has arity (2,1), while the right hand side has arity (3,2). Furthermore, although the tie-point t3 occurs as input on both sides of the overlay, this is not true of t6 and any tie-point of the *bump+update* plan. It does not even correspond to the compound object (the *upper-segment*) represented by t1 and t2. It corresponds to the *upper-segment* viewed as a *list* (via a function *upper-segment->list*). To cope with these features the basic bottom-up chart parser presented earlier is modified as follows:

1. The rule format is modified to include the left-hand side inputs and outputs, since these may now be distinct from those on the right. Correspondingly, each patch now

has two extra fields — *left-hand-ins*, and *left-hand-outs*, in addition to the two fields *inputs* and *outputs* (corresponding to inputs and outputs of the right hand side of the rule). Complete patches are stored in the chart indexed by their *left-hand-ins* and *left-hand-outs*, rather than by their inputs and outputs as before. Partial patches are stored as before.

2. A second chart is added. This chart (the *tie-point chart*) stores the functional relationships between tie-points discovered during parsing. It contains entries with the form:

$$T = F(S_1, \ldots, S_k)$$

where T and $S_1, \ldots, S_k$ are known (i.e. instantiated) tie-points. This chart is organised similarly to the earlier (complete) chart, in that it is split into two parts, one used for storing relationships indexed by the left hand side tie-point (T) and by the relationship name (F), and the other used for storing the relationships indexed by the right-hand side tie-points ($S_1, \ldots, S_k$), and by the relationship name.

3. Two new fields are added to each patch. These are:

   (a) *relations-needed* — When an empty patch is created this is initialised to the set of tie-point relationships of the rule involved in creating the patch.

   (b) *relations-found* — When an empty patch is created this is initialised to empty.

4. When a new patch is created, either by extending a partial patch, or by creating a new empty patch for some rule, any instantiations for the tie-point variables occurring in the patch are also propagated into the relations-needed entry. The following is then repeated until there is no change to the patch:

   (a) If some relationship in the relations-needed entry is fully instantiated (i.e. no tie-point *variables* occur on either its left or right hand sides) then it is moved from the relations-needed entry to relations-found.

   (b) If some relationship in the relations-needed entry has a fully instantiated right-hand side i.e. is of the form:

$$V = F(S_1, \ldots, S_k)$$

   where V is a tie-point variable, and $S_1, \ldots, S_k$ are all known tie- points, then the tie-point chart is consulted to see if there is an entry of the form:

$$T = F(S_1, \ldots, S_k)$$

   where T must be a known tie-point. If there is, then V is instantiated to T, and this instantiation is propagated throughout the patch (including its relations-needed entry). If not, then a *new* tie-point T is *created*, the assertion:

$$T = F(S_1, \ldots, S_k)$$

   is added to the tie-point chart, V is instantiated to T, and this instantiation is propagated throughout the patch (including its relations-needed entry).

   (c) If some relationship in the relations-needed entry has an instantiated left-hand side i.e. is of the form:

$$S = F(T_1, \ldots, T_k)$$

   where S is a known tie-point, and $T_1, \ldots, T_k$ are *either* known tie-points *or* tie-point variables, then the tie-point chart is consulted to see if there is an entry of the form:

$$S = F(S_1, \ldots, S_k)$$

   where $S_1, \ldots, S_k$ are all known tie-points. Matching $T_1, \ldots, T_k$ against $S_1, \ldots, S_k$ either succeeds, in which case any variables in $T_1, \ldots, T_k$ get instantiated, and these instantiations are propagated throughout the patch. If the match fails (because two *different* known tie- points are being matched against each other) then the whole patch is invalid, and is rejected (i.e. is not added to the chart).

5. A patch is only considered complete if *both* its needed entry *and* its relations-needed entry are empty. If they are, then the patch is added to the chart as normal. If not, then the patch is considered partial, and is stored in the chart indexed as before, but also indexed by the relationship names and instantiated tie-points of any immediately needed (in the obvious generalised sense of the term) relationships in the relations-needed entry.

6. When a relationship is added to the tie-point chart, the (patch) chart is consulted to see if there are any partial patches waiting for a tie- point relationship compatible with the one just added. If so, the patch is extended by the relationship, and added to the agenda.

Figure 6 illustrates this for the above rules.

Now consider rules like those in Figure 7, which includes a rule (for A) with a "straight-through" arc, and a graph like that in Figure 8. This can be recognised as forming an S, by the following sequence of events:

1. The NAPEs labelled b and c in Figure 8 are recognised as forming a partial A, which still has:

$$t1 = iterator(1, t4)$$

$$t2 = iterator(3, t4)$$

in its relations-needed entry.

2. The NAPEs labelled d and e in Figure 8 are recognised as forming a complete B, with input given by a em new tie-point 8, and output given by a *new* tie-point 9, satisfying:

$$8 = iterator(3, 4)$$

$$9 = iterator(6, 7)$$

These relationships are added to the tie-point chart.

3. When the relationship 8=iterator(3, 4) is added to the tie- point chart, the main chart is consulted to see if there are any partial patches immediately needing a relationship matching this one. The partial A discovered earlier is found, and on matching

$$8 = iterator(3, 4)$$

and

$$t2 = iterator(3, t4)$$

t4 gets instantiated to 4, and t2 gets instantiated to 8. The patch is therefore extended, and it now has only the single relationship:

$$t1 = iterator(1, 4)$$

in its relations-needed entry. This causes the creation of a new tie-point 10 to which t1 is instantiated, and an assertion:

$$10 = iterator(1, 4)$$

is added to the tie-point chart. As a result of all this we now have a complete A patch (with input 10 and output 8) which gets added to the chart. This causes the creation of an empty S patch (with input 10) immediately needing an A (with input 10) to be added to the chart.

4. This patch is extended first by the A patch, and then again by the B patch, giving us a complete S patch with input 10 and output 9, where:

$$10 = iterator(1, 4)$$

and

$$9 = iterator(6, 7)$$

This illustrates very nicely the role of the second chart.

7 Chart Parsing of Structure-Sharing Flowgraphs

As stated earlier we are also interested in the case where structure sharing is allowed. However, for reasons discussed in Lutz (1992), we do not want to allow *any* two NAPEs sharing the same inputs to be collapsed, but only NAPEs with appropriate labels. To make this more precise we define a slightly more general notion:

A *restricted structure sharing flowgraph grammar* (RSSFG) is a 5-tuple (N, T, P, S, R) where N, T, P, S are the same as for ordinary context

free flowgraph grammars, and $R \subseteq N \cup T$. R is the set of NAPEs for which collapsing is allowed. Such a grammar has an additional rewriting rule which will be described below. We define a relation *R-collapses* on the set of flowgraphs over $N \cup T$ by:

> G2 *R-collapses* G1 iff G1 and G2 are flowgraphs, and $G1^c$ contains two triples of the form $T1 = (A,(t_1,\ldots,t_n),(x_1,\ldots,x_m))$ and $T2 = (A,(t_1,\ldots,t_n),(y_1,\ldots,y_m))$, where A $\in R$ and $G2^c$ can be obtained from $G1^c$ by removing these two triples and replacing them by a single triple of the form $T3 = (A,(t_1,\ldots,t_n),(z_1,\ldots,z_m))$ and then replacing all occurrences of $x_1,\ldots,x_m$ and $y_1,\ldots,y_m$ by $z_1,\ldots,z_m$ respectively throughout the remaining triples.
>
> In other words, G2 *R-collapses* G1 iff G1 contains two instances of some NAPE A (whose label is in R) which have the same inputs, and G2 is identical to G1 except that the two instances of A have been replaced by a single instance of A (with the same inputs) and all NAPES which originally connected to the outputs of one or other of the two instances of A now connect to the single instance (in G2). This amounts to identifying the two instances of A and their corresponding tie-points.

The reflexive, transitive, symmetric closure of *R-collapses* is then an equivalence relation (*R*-share-equivalence) on the set of flowgraphs, and we want any parsing algorithm which can recognise some graph G to also be able to recognise any flowgraphs R-share-equivalent to G. We also want the grammar to be able to generate not only the flowgraphs derivable directly from the grammar, but also all R-share-equivalent flowgraphs. This can be done if we allow at any point in the generation of a flowgraph the replacement of the graph so far generated (G1) by any graph G2 for which *either* G1 *R-collapses* G2 or G2 *R-collapses* G1. If $R = \emptyset$ then the grammar is an ordinary flowgraph grammar, and if $R = N \cup T$ then we get a (full) structure sharing flowgraph grammar as defined in Lutz (1989). Figure 9 illustrates several phenomena that can occur with RSSFGs, and which motivated the above definition.

To see how the parsing algorithm can be modified to cope with RSSFGs it should first be noted that for any flowgraph G there is a *smallest* flowgraph G_{min} which is R-share-equivalent to G. Secondly, the right-side structure of any rule in an RSSFG can be replaced by any flowgraph R-share-equivalent to it without altering the generative capacity of the grammar. We therefore define a canonical form for an RSSFG in which each rule of the form:

$$A ==> B$$

has been replaced by the rule:

$$A ==> B_{min}$$

So the first change to the algorithm is to put the grammar into canonical form. The second change is to the action of adding a complete patch to the chart. Previously the only check that was done was to see if the patch was already in the chart. Now the algorithm must additionally check if the label of the patch is in R and if there is another patch in the chart with the same label and the same inputs. If so, the algorithm must collapse the new patch and the one already present into a single patch, by identifying the output tie-points of the two patches. Provided tie- points in the various triples making up the patches are represented as pointers to pointers to tie-points (rather than storing the tie-points directly in the triples) then simply changing the values of the second set of pointers will implement the identification universally throughout all patches in the chart. This can lead to "chains" of pointers which need to be fully dereferenced in order to actually access the tie-points themselves (this is similar to the way variables are handled in many implementations of Prolog). After collapsing an additional step is needed since there may be patches in the chart indexed by the tie-point which has been effectively removed by the identification. These patches must now be stored in the chart indexed by their new output tie-points. If the information that collapsing has been done is needed by an application the algorithm can make a note this fact either by annotating the tie- points involved

or by an assertion held separately. Finally, the no-sharing check must be omitted.

If the grammar also has attributes, then we need to specify how to compute the attribute A_{res} of a NAPE resulting from collapsing two patches with attributes A_1 and A_2. This specification takes the form (for each attribute):

$$A_{res} = f_{collapse}(A_1, A_2)$$

where $f_{collapse}$ is a function which computes the value of the attribute for the new patch from the values for the two collapsed patches. Two NAPEs are only collapsed when an attempt is made to add a complete patch P_2 (with attribute A_2) to the chart, and there is already a patch P_1 (with attribute A_1) present in the chart with the same label and inputs. In this case P_1 is left in the chart (i.e. it is the output tie-points in P_2 which are identified with those in P_1). P_1 then has the value of its attribute set to $f_{collapse}(A_1, A_2)$. If this new value for its attribute is different from its previous value, then any complete patches in the chart which have P_1 as one of their components must also have their attributes recalculated, and any of these patches whose attributes change must also have their attributes recalculated, and so on recursively. To facilitate this, each complete patch P needs an extra field *partof* which holds a list of all complete patches of which P is a component. To maintain this field, whenever a complete patch P is added to the chart, P is added to the *partof* field of each of its component patches. The initial patches (corresponding to the original graph) all have this field initially set to empty.

8 Applications

The algorithm just described forms the basis of the program understanding process described in Lutz (1989b, 1991, 1992). However, there are other domains, in particular digital circuit analysis, in which a similar ability to parse flowgraphs is useful. Consider Figure 10 which shows a circuit for addition of 3-bit numbers. The grammar shown in Figure 11 is capable of generating this circuit. Adding a rule like that shown in Figure 12 enables the parser to recognise the circuit in Figure 10 as being equivalent to Figure 13 i.e. to recognise the circuit as adding two numbers, with the tie-point chart holding information about how the numbers have been "implemented".

9 Conclusions

This paper has presented a polynomial time chart parsing algorithm for context-free flowgraph languages, capable of handling all the features of the plan calculus (Rich, 1981), and which is also applicable to digital circuit analysis.

Although there is a large literature on the generative abilities of various types of graph grammar formalisms (see e.g. (Ehrig, 1979; Feder, 1971; Fu, 1974; Gonzalez — Thomason, 1978; Pfaltz — Rosenfeld, 1969; Rosenfeld — Milgram, 1972)), there is relatively little on parsing strategies, except for restricted classes of graph and web grammars (e.g. Della Vigna — Ghezzi (1978)). In its top-down strictly left-to-right form chart parsing of context-free string languages corresponds to Earley's algorithm (Earley, 1970), which was generalised by Brotsky (1984) to parsing flowgraphs of the kind described here, except that his algorithm could not cope with fan-out at tie-points, or with tie-point relationships. However the approach taken here can also run bottom-up, which is particularly useful in applications in which we want to recognise as much as possible even though full recognition may be impossible (because of errors in the graph, or because the grammar is necessarily incomplete). Wills (1986, 1990) modified Brotsky's algorithm to cope with fan-out, but her algorithm only runs in a pseudo-bottom-up fashion by starting it running top-down looking for every possible non-terminal at every possible place in the graph. More recently, Wills (1992) developed an algorithm heavily based on the chart parsing algorithm described here and in (Lutz, 1986, 1989), which is also capable of dealing with attributes and tie-point relationships. However her algorithm and graph representations make no mention of tie-points, but deal directly with the edges connecting NAPEs. This makes it harder to deal elegantly with the tie-point relationships.

More recently there have been several papers in the visual language literature which have adopted a chart parsing approach. In particular, Wittenburg et al. (1991) and Golin (1991) have both developed bottom up parsers for 2-

dimensional languages, while O'Gorman (1992), Costagliola et al. (1991), and Wittenburg (1992) have developed top-down parsers. This work is all similar in spirit to that presented here, although differences in representation, and application, make it very different in detail. Indeed this seems to be a general problem with work on 2-dimensional languages - there is no known general method suitable for conveniently representing all the different classes of language, and this leads to algorithms for one domain being very different from those in another. Of course, some kind of definite clause encoding could be used for all of these, but this is not always natural, and does not always lend itself to the development of efficient algorithms. In this connection it should be noted that the flowgraph languages discussed in this paper can be encoded in the Datalog formalism (Abitoul — Vianu, 1988) for which it is known that parsing can be performed in polynomial time. However, a special-purpose algorithm like the one presented can be particularly efficient and adaptable (cf. the control of structure sharing).

A particular advantage of a chart parser is that it keeps a record of all partial patches. This is useful when we do not just wish to analyse how some graph has been generated, but also to make suggestions based on "near-miss" information about how to correct the graph. As such this algorithm is being used as the basis of an intelligent debugging system for Pascal programs (Lutz, 1992).

The algorithms described in this paper have been implemented in Pop-11, running under the POPLOG™ system.

References

Abitoul, S. — V. Vianu (1988) "Datalog Extensions for Database Updates and Queries". I.N.R.I.A. Technical Report No. 715

Brotsky, D.C. (1984) "An Algorithm for Parsing Flow Graphs". AI-TR-704. MIT Artificial Intelligence Laboratory.

Costagliola, G. — M. Tomita — S.-K. Chang (1991) "A Generalised Parser for 2-D Languages" In: *Proceedings of IEEE Workshop on Visual Languages* 98 - 104.

Della Vigna, P. — C. Ghezzi (1978) "Context Free Graph Grammars". In: *Information and Control* Volume(37), 207 - 233.

Earley, J. (1970) "An Efficient Context-Free Parsing Algorithm". In: *Communications of the ACM* Volume(13), 94 - 102.

Ehrig, H. (1979) "Introduction to the Algebraic Theory of Graph Grammars (A Survey)". In: Claus, V. & H. Ehrig & G. Rozenberg, (Eds): *Graph Grammars and their Application to Computer Science and Biology* Lecture Notes in Computer Science. Springer-Verlag.

Feder, J. (1971) "Plex Languages". In: *Information Sciences* Volume(3) 225 - 241.

Fu, K.S. (1974) *Syntactic Methods in Pattern Recognition* New York: Academic Press.

Golin, E.J. (1991) "Parsing Visual Languages with Picture Layout Grammars" In: *Journal of Visual Languages and Computing* Volume(2), 371 - 393.

Gonzalez, R.C. — M.G. Thomason (1978) *Syntactic Pattern Recognition: An Introduction* Addison-Wesley.

Lutz, R.K. (1986) "Diagram Parsing - A New Technique for Artificial Intelligence". CSRP-054. School of Cognitive and Computing Sciences, University of Sussex.

Lutz, R.K. (1989a) "Chart Parsing of Flowgraphs". In: *Proceedings of 11th Joint International Conference on AI, Detroit, USA*

Lutz, R.K. (1989b) "Debugging Pascal Programs Using a Flowgraph Chart Parser". In: Proceedings of 2nd Scandinavian conference on AI, Tampere, Finland.

Lutz, R.K. (1991) "Plan Diagrams as the Basis for Understanding and Debugging Pascal Programs". In: Eisenstadt, M. & T. Rajan & M. Keane, (Eds) *Novice Programming Environments* London: Lawrence Erlbaum Associates.

Lutz, R.K. (1992) "Towards an Intelligent Debugging System for Pascal Programs: On the Theory and Algorithms of Plan Recognition in Rich's Plan Calculus". Ph.D. Thesis The Open University, Milton Keynes, England.

O'Gorman, L. (1992) "Image and Document Processing Techniques for the RightPages Electronic Library System" In: *Proceedings 11th IAPR International Conference on Pattern Recognition* Volume(2), 260 - 263.

Pfaltz, J.L. — A. Rosenfeld (1969) "Web Grammars". In: *Proceedings of 1st International Joint Conference on AI* 609 - 619.

Rich, C. (1981) "Inspection Methods in Programming". AI-TR-604 MIT Artificial Intelligence Laboratory.

Rosenfeld, A. — D.L. Milgram (1972) "Web Automata and Web Grammars". In: Meltzer, B. & D. Michie (Eds): *Machine Intelligence* 7, 307 - 324. Edinburgh University Press.

Thompson, H. — G. Ritchie (1984) "Implementing Natural Language Parsers". In: O'Shea, T. & M. Eisenstadt (Eds) *Artificial Intelligence: Tools, Techniques, and Applications* 245 - 300 Harper and Row.

Wills, L.M. (1986) "Automated Program Recognition". MSc Thesis. MIT Electrical Engineering and Computer Science.

Wills, L.M. (1990) "Automated Program Recognition: A Feasibility Demonstration". In: *Artificial Intelligence* Volume(45), 113 - 171.

Wills, L.M. (1992) "Automated Program Recognition by Graph Parsing". Ph.D. Thesis. MIT, Boston, Mass.

Wittenburg, K. (1992) "Earley-style Parsing for Relational Languages" In: *Proceedings of IEEE Workshop on Visual Languages* 192 – 199.

Wittenburg, K. — L. Weitzman — J. Talley (1991) "Unification-based Grammars and Tabular Parsing for Graphical Languages" In: *Journal of Visual Languages and Computing* Volume(2), 347 – 370.

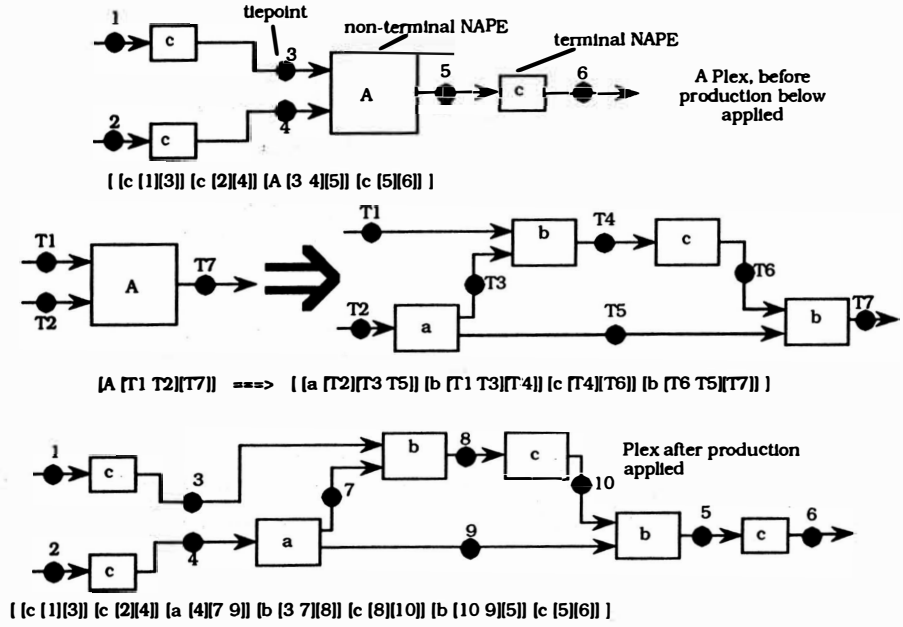


Figure 1.
Simple Flowgraph and Rules

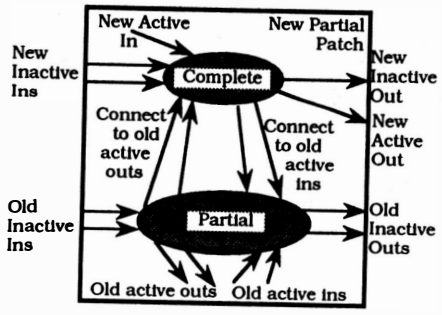
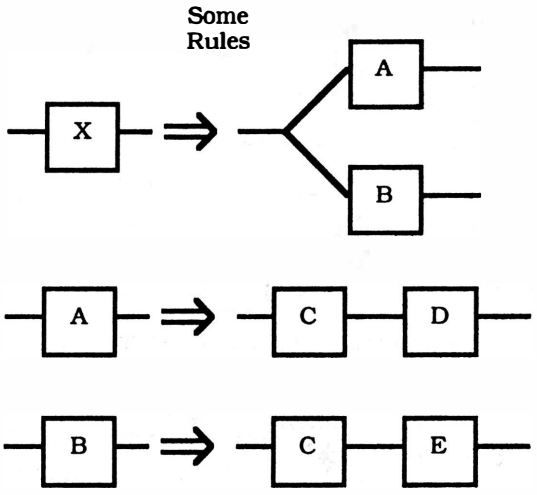


Figure 2
The Joining Operation



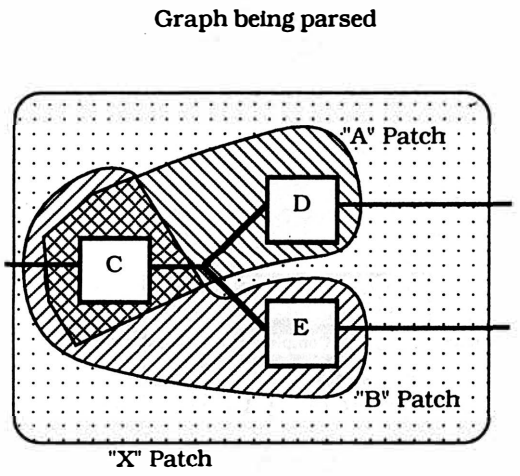
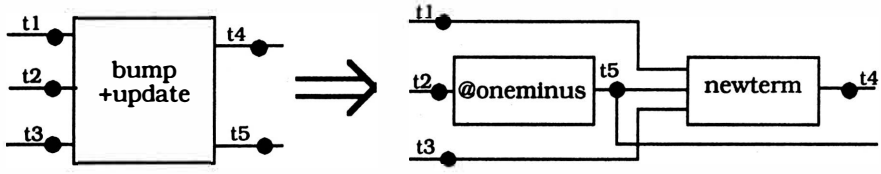
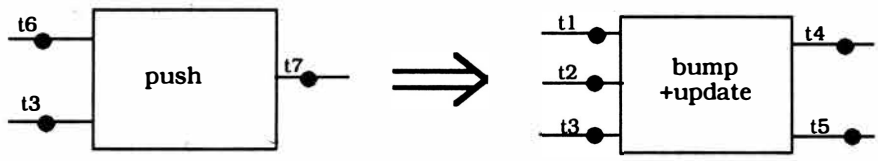


Figure 3
Occurence of Structure Sharing Without No Sharing Check



where t6=upper-segment(t1,t2) and
t7=upper-segment(t4,t5)

Figure 4
Bump+update



where
t6=upper-segment->list(t8) and
t7=upper-segment->list(t9) and
t8=upper-segment(t1,t2) and
t9=upper-segment(t4,t5)

Figure 5
Bump+update->push

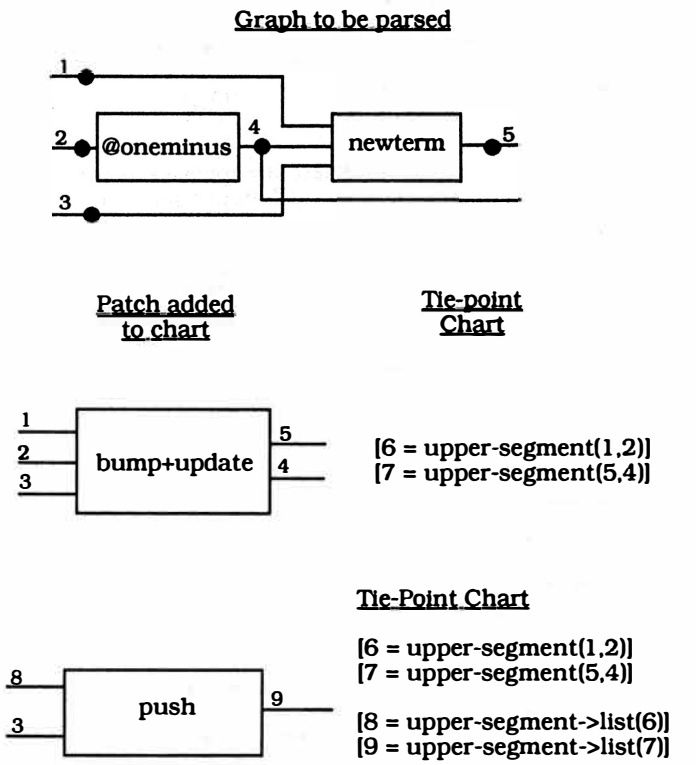
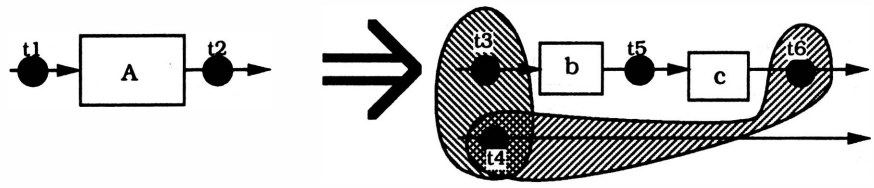
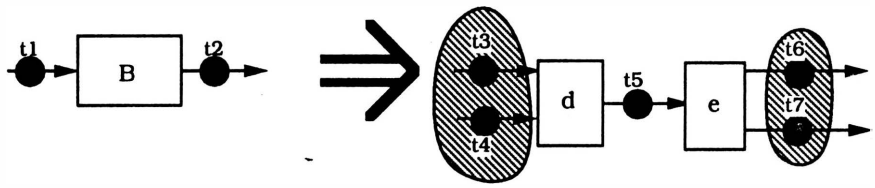


Figure 6
Use of Tie-point Chart



where t1=iterator(t3, t4)
and t2=iterator(t6, t4)



where t1=iterator(t3, t4)
and t2=iterator(t6, t7)

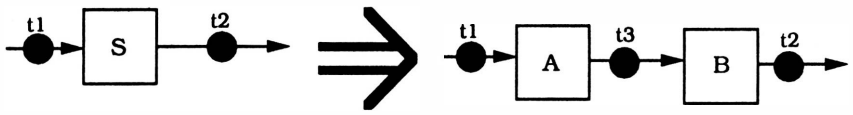


Figure 7
Some Rules (with a "straight-through" arc)

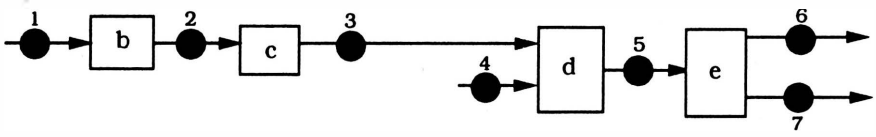


Figure 8
Graph to be Parsed

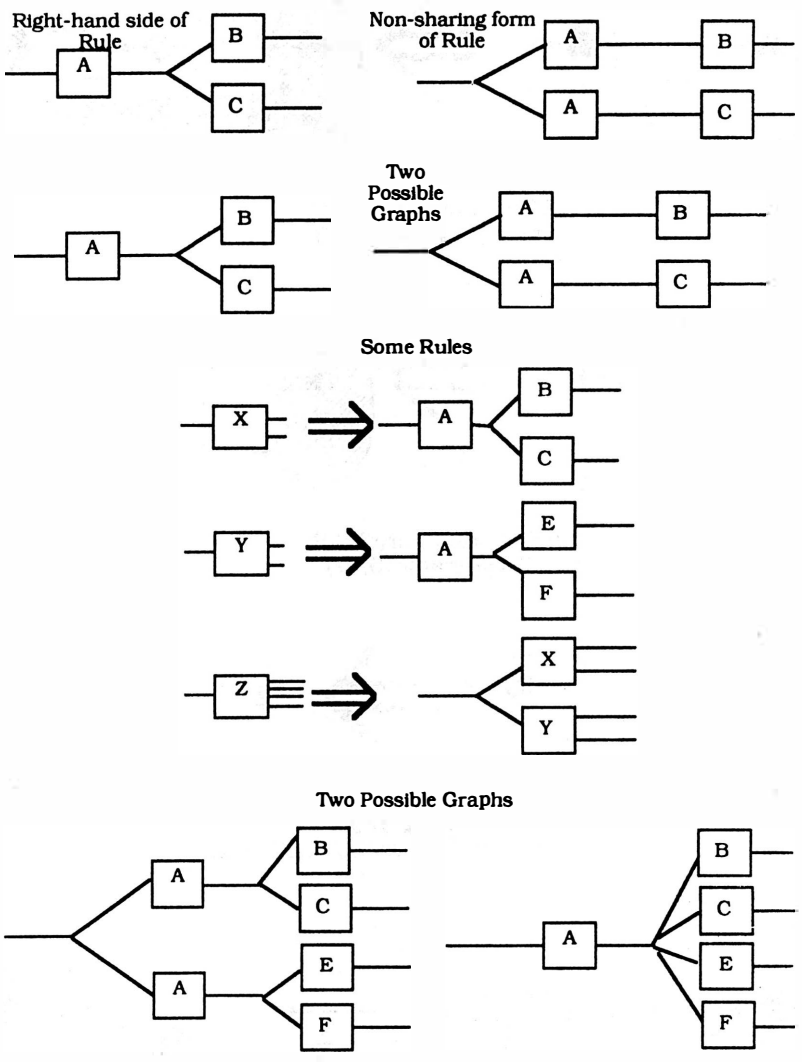


Figure 9
Structure Sharing and Collapsing Phenomena

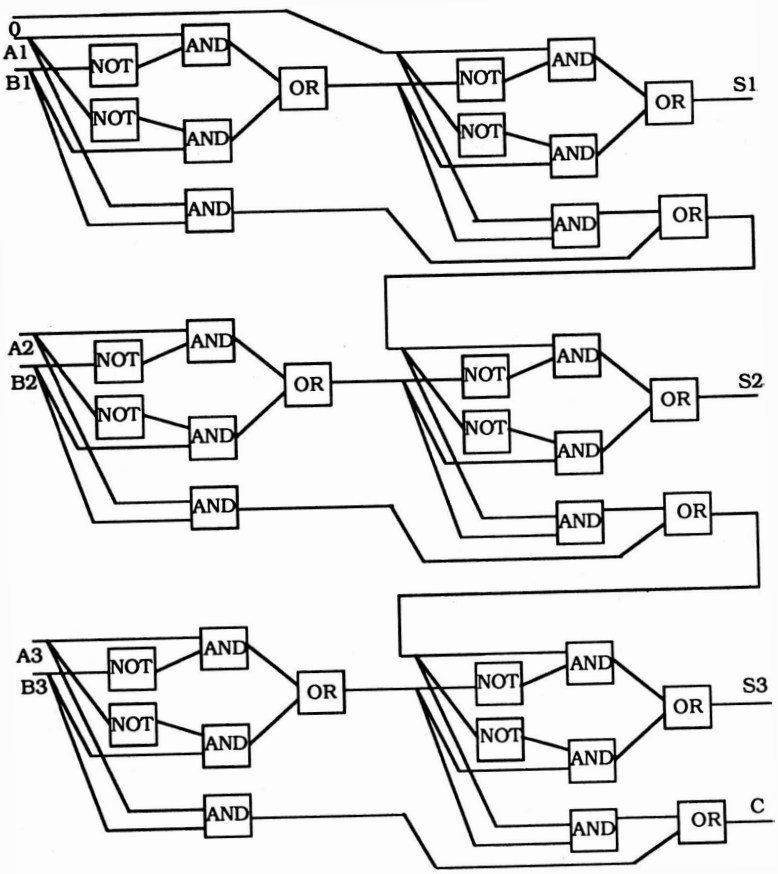


Figure 10
A 3-Bit Addition Circuit

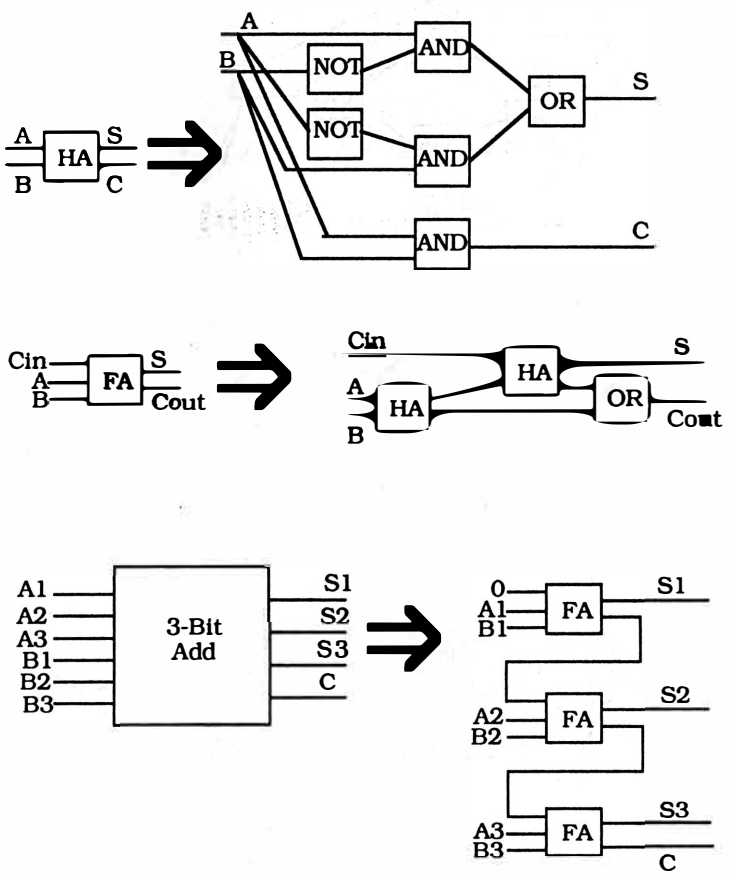
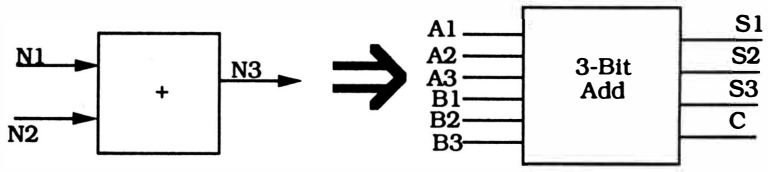
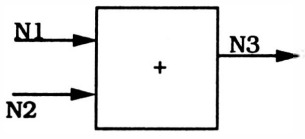


Figure 11
Addition Circuit Grammar



where
Binary1=3-bits(A1,A2,A3) and
Binary2=3-bits(B1,B2,B3) and
Binary3=4-bits(S1,S2,S3,C) and
N1=3-bits->integer(Binary1) and
N2=3-bits->integer(Binary2) and
N3=4-bits->integer(Binary3)

Figure 12
Integer Addition Rule



where
Binary1=3-bits(A1,A2,A3) and
Binary2=3-bits(B1,B2,B3) and
Binary3=4-bits(S1,S2,S3,C) and
N1=3-bits->integer(Binary1) and
N2=3-bits->integer(Binary2) and
N3=4-bits->integer(Binary3)

Figure 13
High-Level Description of Figure 10

The Interplay of Syntactic and Semantic Node Labels in Partial Parsing

David D. McDonald

Dept. of Computer Science, Brandeis University
14 Brantwood Road, Arlington MA 02174, USA
email: mcdonald@chaos.cs.brandeis.edu

Abstract

Our natural language comprehension system, "Sparser", uses a semantic grammar in conjunction with a domain model that defines the categories and already-known individuals that can be expected in the sublanguages we are studying, the most significant of which to date has been articles from the Wall Street Journal's "Who's News" column. In this paper we describe the systematic use of default syntactic rules in this grammar: an alternative set of labels on consitutents that are used to capture generalities in the semantic interpretation of constructions like the verbal auxiliaries or many adverbials. Syntactic rules form the basis of a set of schemas in a Tree Adjoining Grammar that are used as templates from which to create the primary, semantically labeled rules of the grammar as part of defining the categories in the domain models. This design permits the semantic grammar to be developed on a linguistically principled basis since all the rules must conform to syntactically sound patterns.

1 Partial parsing and semantic grammars

The rate-limiting step in advancing how much machines can understand from a natural language text is the size and thoroughness of the semantic models used to make sense of the information the texts contain. At the same time, the state of the art in parsing and information extraction mandates that the texts being analyzed can no longer be artificial — made up by linguists to test their grammatical theories — but should be actual texts written by people for other people; indeed, the right kind of test corpus today will be downloaded from an online service on the day of the test.

To accomodate this combination of unrestricted sources and limited semantic competence, one focuses on sublanguage analysis (in the sense of Kittredge & Lehrberger 1982). This may be simply a restriction in the source of one's texts to just one topic and register, such as in the present instance single paragraph articles taken from the "Who's News" column of the Wall Street Journal, which contain almost nothing beyond announcements of the promotions or retirements of executives in commercial businesses. It may more broadly be a concentration on a given kind of information and its standard sublanguage but with no constraint on the sources in which it appears, e.g. extracting executive job change information in articles of any length appearing anywhere in the Journal or other business news sources available online, while at the same time functioning robustly dispite the unknown words and unparseable constructions in other portions of the texts.

This then is the sence in which the "partial parsing" of this paper's title should be understood: the parses the system makes, and its comprehension of the parsed segments, should be both complete and thorough, but those segments will typically constitute only a portion of the whole text being analyzed. Since the intrusion of off- topic segments can be at any granularity — apposatives or adjuncts within clauses,

modifiers within NPs, etc., a partial parser in this sense will incorporate some heuristic techniques for compensating for such gaps in its analysis, but for the most part its parsing machinery can be conventional and its only substantial differences from a parser intended to always parse every part of its text will lie in its grammar.

In order to constrain the grammar to only those topics and sublanguages within the corpus that the system's semantic models will understand, one is naturally drawn to the use of semantic grammars. Here the grammar will only provide an analysis if the model will comprehend it — no parse node will be allowed to be formed if it does not also have a semantic interpretation. No effort is wasted in trying to make sense of portions of the text not on the target topic since the grammar will leave them unanalyzed and no interpretations will be initiated.

Introduced originally by Dick Burton as the basis of the natural language interface to the Sophie ICAI system (Burton 1976; Burton & Brown 1975) and later used for the interfaces in Ladder (Hendrix et al. 1978), Planes (Waltz et al. 1976), and many other systems, semantic grammars have the advantage of high specificity, easy, almost common-sense definition, and extremely low run-time ambiguity. At the same time, they also suffer from brittleness, minimal carryover and significant effort in extending them to other domains, and a striking lack of ability to appreciate linguistic generalizations.

In this paper we will describe a version of semantic grammar that addresses these problems while retaining the formalism's strengths. We provide a systematic basis for relating the semantic categories used by the parser to the categories and representational structures used in the semantic model that provides the text's interpretation. We also provide the parser with a parallel set of standard syntactic categories shadowing the semantic categories and providing the basis of domain-independent default rules with a common semantics. This is all brought together in a set of general schemas, derived from a Tree Adjoining Grammar, that allow the definition of the grammar to be done as part of the definition of the semantic model, ensuring consistency and reducing much of the labor of writing semantic grammars in the process.

We will begin by motivating the way in which we link parsing rules to their interpretations, and with that introduce the notion of schemas that are used to define the two simultaneously. We will then describe the 'form' rules that provide the syntactic defaults, and present the algorithm for checking for possible constituent combinations in some detail since it is the means by which the two kinds of rules are interleaved.

2 Parsing to objects

What is the goal of a parser? The customary answer is that it is to determine the form of a text — its decomposition into constituents and the grammatical relations among them, which is to then serve as the basis of its semantic interpretation — the recovery and representation of its meaning. The line between form and interpretation begins to blur, however, once one adopts a semantic grammar as the basis of the analysis of text form. A constituent in a semantic grammar has a label that reflects its category when taken as an thing or event-type in the world: 'person' rather than NP, 'elect' or 'get-position' rather than verb. Given a multi-level analysis by the parser, with transformations or their equivalent to render a surface-level representation into a canonical form, such a choice of categorizations will yield a representation not too disimilar from some conceptions of logical form, with the constituent labels providing the predicate names applied to variables introduced by the transformations.

The present work, however, employs a denotational rather than descriptive notion of semantic interpretation. Here the process of parsing a text incrementally accesses or contructs a set of unique semantic entities — individuals in a model of the world and the ongoing discourse as represented in the mind (code) of the language user. The interpretation function is applied rule by rule as the parse proceeds, rather than having to wait for it to end. Many of the individuals and all of the base types of this model will exist before the text is processed, and the task of the language understanding system is taken to be to recover from its analysis of the text the objects representing these individuals, and to add to the model any newly established individuals and any new relations or properties over them and pre-existing individuals.

As a parser then, the Sparser system does re-

cover a phrase structure-based analysis of a text's form. But this form is only a means to an end, namely to project from the words of the text, as channeled by the constituent structure, directly to the objects in the world/discourse model without any interveening intermediate level of representation of the meaning as another expression such as a logical form. Since this projection happens incrementally as the parse proceeds from beginning to end in one pass through the text, the parser may as well be said to be parsing to objects (in something of the manner of Reisbeck and Martin 1985), since it will not complete a constituent without also recovering that constituent's counterpart in the model.

This being the case, and given the goal of a partial parser to tightly coordinate what the parser's grammar is able to analyze with what its semantic model is able to understand, a natural way to proceed is to have the rules of the grammar written as a side-effect of defining the types of the model. What this means is that when one defines, say, a category of event like 'confirm-in-position', that definition should include the definition of the rules needed to parse texts about that event-type. (Confirm-in-position is intended here as the denotation of the verb "confirm" as used in a sentence like "*J. Gordon Strasser, acting president and chief executive officer of this gold mining company, was confirmed in the posts* ...", which is typical of the "Who's News" texts we have worked with.) Concombinantly, the parsing rules created by this definition should spell out not only how the constituents of the text compose, but also how those compositions should be projected to the discourse model to incrementally construct an individual to represent the confirmation event rule by rule as the parse progresses.

Let us consider what has to go into these definitions so that establishing 'confirm-in-position' as an event-type in the model will at the same time provide the rules that the grammar needs to handle a sentence like the example. For its own purposes, the semantic definition of an event or other relations will specify a set of necessary and optional participants, including value restrictions constraining what categories of individuals those participants can be. For 'confirm-in-position', as evidenced by news articles in the Wall Street Journal, the agent that causes a confirmation can be either a company or its board of directors; the participant who gets confirmed is a person; and that person is confirmed to some position. We can use these categories directly to provide the labels for the corresponding constituents in the semantic grammar, where they take on the force of selectional restrictions.

Beyond the selectional restrictions on the participants, the definitions must supply the words that have the intended meaning as one of their senses (i.e. the semantic object under definition) as well as the syntactic subcategorization frames that the words take. Given the choices of lexical realization, syntactic information is then needed to spell out the phrasal patterns corresponding to the subcategorizations and how the participants project onto them.

With few exceptions, the subcategorization frames are shared by a great many words, and consequently can be schematized and shared among the definitions (e.g. transitive verb, intransitive, noun taking quantification of number, noun holding a position in a sequence, etc.). The patterns of verbs especially are most naturally given in terms of syntactic phrasal categories, and can thus also provide the default syntactic categories that are provided with the rules of the semantic grammar, as we shall see.

We fill the requirement to fit all of the participants of a relation to a single sharable syntactic schema by adopting Lexicalized Tree Adjoining Grammars as the linguistic formalism (Joshi 1985; Shabes, Abeille & Joshi 1988). TAG's extended domain of locality allows the configurational and grammatical relations of all of a lexical head's arguments to be given at once, which is important here since the semantic relationship of each argument to its matrix relation must be specified in the schema, and this is easier to do if all of the participants of a domain relation can be paired with their syntactic counterparts in one statement. The schemas we adopt are a notational variant on the TAG Tree Families of Abeille 1988, which give all of the transformational variations of a subcategorization frame in one structure. The particulars are described in section five.

To summarize the framework so far, we are employing in the Sparser system a grammar of rules of syntactic form which employ semantic categories as their primary labels, with standard syntactic labels provided with them as defaults. Each rule has an accompanying semantic inter-

```
(1)   job-event  ->  board-of-directors  job-event\agent
(2)   job-event  ->  company  job-event\agent
(3)   job-event  ->  job-event  in-position
(4)   job-event  ->  job-event  as-position
(5)   job-event  ->  person  confirm-in-position+passive
(6)    job-event  ->  person+pos  confirm-in-position+nominalization
(7)   job-event  ->  confirm-in-position+nominalization  of-person
(8)   job-event\agent  ->  confirm-in-position  person
(9)  confirm-in-position  ->  "confirm"
(10) confirm-in-position  ->  "confirms"
(11) confirm-in-position  ->  "confirmed"
(12) confirm-in-position  ->  "confirming"
(13) confirm-in-position+nominalization  ->  "confirmation"
```

Figure 1: Semantically labeled rules

pretation function that will find or construct the denotation of the constituent it creates. The grammar rules and the set of model-level object types are created at the same time by definitions that recruit the value restrictions on the participants in a relation to supply the semantic labels on the grammar's rules, where they implicitly serve the role of selectional restrictions ensuring that the rules cannot complete unless their constituents will make sense to the world model. The set of rules created by a type definition is determined by instantiating the TAG tree family(s) corresponding to the subcategorization frames of the word or words that can realize the relation-type in a text.

The effect on the final grammar is as though we had taken a conventional syntactic grammar that used syntactic preterminal categories and performed two multiplications: the first takes its preterminal categories for content words and multiplies them across all of the lexical choices specified by the definitions of the model's domain-specific types. The second takes the rules composing normal syntactic nonterminals and multiplies them across all of the sets of selectional restrictions dictated by the semantic categories of the individuals that can participate in the domain relations. The resulting grammar is dramatically larger than the original, but it will show a equally dramatic increase in runtime efficiency because of its minimal ambiguity and its immediate and incremental connection to the semantic interpretation, which, after all, is the ultimate purpose of the grammar in the first place.

3 Integrating rewrite rules and denotations

Before introducing the notation and mechanisms of the model-level definitions and the subcategorization schemas, we should look at some examples of the grammar rules — productions — that Sparser uses. We will start with some semantically labeled rules based on a type definition — the output of a schema, and then look at the syntactically labeled rules that complement it. We start with the set of rules produced for 'confirm-in-position', all of which are listed just in terms of their semantic labels in figure 1 We will then look at the details of what is represented in one of these rules and show how the syntactic and denotational information is represented and put to use. We will end the section by sketching how the rules are used in a parse of a text that is completely within the job-change sublanguage so that one can get a sense of the interplay between form and denotation in the Sparser system.

```
#<phrase-structure-rule psr11
  :lefthand-side  #<category confirm-in-position>
  :righthand-side  ( #<word ''confirmed''> )
  :syntactic-form  #<category verb+ed>
  :referent  #<category confirm-in-position> >
```

Figure 2: Rewrite rule 11

```
#<edge   e22 ;; a designator to use as the edge's name
  :starts-at #<position 18>
  :ends-at #<position 19>
  :category #<category confirm-in-position>
  :left-daughter #<word ''confirmed''>
  :right-daughter :unary-rule   ;; i.e. only the 'left' daughter matters
  :used-in nil  ;; later points to the edge this is a daughter of
  :form #<category verb+ed>
  :referent  #<category confirm-in-position> >
```

Figure 3: An edge in the chart

This rule set (Figure 1) consists of five unary lexical rules (9 - 13), and eight binary rules over non-terminals corresponding to the conventional rules: S -> NP VP, VP -> VG NP, S -> NP VG, S -> S PP, and two NP rules for nominalizations. These semantically labeled rules are complemented by a set of generically applicable syntactically labeled 'form rules', e.g. for the "be + ed" of the passive, other auxiliaries and standard adverbs, and for the definite or indefinite articles that can occur with the nominalizations. We take these up in a moment.[1]

In parsing our example, "*J. Gordon Strasser, acting president and chief executive officer of this gold mining company, was confirmed in the posts*", the instantiation of a model-level 'confirm-in-position' event will start when the word "*confirmed*" is scanned, firing rewrite rule 11 (figure 2). That rule has the internal representation shown below. The '#<>' notation indicates pointers to objects, with the first term in the brackets indicating the object's type and the rest a summarizing print form, typically the symbol or string used to name it.

Note that the 'referent' given with the rule is a pointer directly into the system's semantic model. It is available as part of the statement of the rule because the rule is created as a side-effect of defining that event-type. Notice also that the rule also indicates the default syntactic label (category) that is to be given to the constituents it forms, i.e. verb+ed, complementing the primary semantic label given as the rule's lefthand side.

When completed, this rule creates a nonterminal node — an edge in Sparser's chart — as shown in figure 3. For purposes of illustration we have the example text start at chart position zero, which puts the word "*confirmed*" between positions 18 and 19. (In the actual text, this was the first clause of the first sentence of a one paragraph article, with an extensive set of headers and with the name and location of the company where the change was going on announced in a dateline just

[1]There are not as many phrasal rules in this set as one might expect for a transitive verb given the notion of a transformation family. This is because Sparser handles the relative pronouns of relative clauses by rewriting them with the semantic label of their NP head, making them equivalent here to rules 2 and 5, and it treats other WH constructions using 'form rules' as described below. Other predictable constructions such as it-clefts or topicalization have not yet occured anywhere in the studied portions of the Who's News corpus and have been omitted just because we have not had the examples to study for their semantic analysis.

before the sentence started — all of which was also entered into the chart and processed.)

Note that the edge essentially just carries over the information given in the rule. The category given as the 'lefthand-side' of the rule becomes the edge's 'category' — it is this semantic label which will be the primary label used the course of parsing. The category given as the 'syntactic-form' of the rule becomes the edge's 'form' — its syntactic label, which will be used only if the semantic label cannot participate in a composition with any of the neighbor edges, i.e. it is the default.

The semantic analysis at work here holds that content words (nouns, verbs, adjectives) name categories of individuals in the semantic model. The denotation of "*confirmed*" is, as shown, the category ascribed to events of that type (which we also use as the semantic label on its edge, a common but not universal practice in this grammar since particularities of subcategorization and applicable substitution contexts can override it and call for more specific labelings). The denotation of the word "*company*" in the example is the category used for all individuals representing companies and similar enterprises; "*posts*" denotes the category for positions, specialized in this case to note that more than one title and/or company is involved; and so on.

The composition of modifiers onto a head constituent may specialize the category of the head to one of its subcategories (as done implicitly with the plurality of "*posts*"), or may predicate attributes to an individual. The composition of some discourse anchor — tense for verbs or determiners for nouns — will change the denotation to now pick out some individual fitting the category of the phrase accumulated on the headline to that point. This happens when "*was*" and "*confirm*" are composed, where we will establish an individual of domain-type (category) 'confirm-in-position'. This involves a 'rule of form' which we take up in the next section.

In the interests of space we will jump ahead to the point when the major constituents of the example have been formed so that we can concentrate on how the rules for confirm-in-position are applied and how they take the new individual in the model — an instance of this event-type — and fill its participant roles with the individuals denoted by those constituents.

Below (figure 4) is a vertical presentation of that chart state, followed by a list of the edges with their semantic and syntactic labels and their denotations. A compact notation is used here for illustration purposes. The edges names, e.g., 'e23', reflect the order in which the edges were formed. The numbers around the excerpted texts are the chart positions.

At this point SPARSER will apply rule 5 from the set above to combine edges e20 and e23 to form a clause ('S') labeled 'job-event' from position 1 to position 20, edge e30. When that edge is formed, the denotations of its daughter edges are combined following the interpretation function given with the rule (see section 5). The result, the denotation of e30, will be the same individual denoted by e23, but now that individual is augmented by having its 'agent' role bound to the person pointed to by the subject edge e20.

```
e20     1 J. Gordon Strasser I of this gold mining company 18
e23     18 was confirmed 20
e29     20 in the posts 23

[e20: person, np, #<individual:person Strasser> ]

[e23: confirm-in-position+passive, vg, #<individual:confirm-in-position> ]

[e29: in-position, pp, #<individual:position:collective President I> ]
```

Figure 4: Chart state and list of edges

```
#<edge  e31
  :starts-at #<position 1>
  :ends-at #<position 23>
  :category #<category job-event
  :left-daughter #<edge 30>
  :right-daughter #<edge 29>
  :used-in nil
  :form #<category S>
  :referent
    #<individual
      :type ( confirm-in-position past get-position job-event
              transition-event )
      :bindings ( #<binding person = #<individual:person Strasser>>
                  #<binding position =
                        #<individual:position:collective President I>> )>>
```

Figure 5: Edge e31

This new edge, e30, will then be combined with the pp-adjunct edge e29 using rule 3 with a similar result. Another edge, e31, is formed as shown in figure 5. Its referent is again the same individual that was instantiated when the verb group edge was formed (i.e. the individual representing this instance of confirm-in-position). Now it gets its 'position' role bound to the individual of type position denoted by e29.

The parse is now finished syntactically since we have recovered a grammatically complete unit, a sentence. Semantically it is still incomplete, since the text did not explicitly give the agent that did the confirming and so the event-type is not yet saturated. In this genre it would be safe to infer that the agent is the company where the person now holds the position.

4 Form rules

The preponderance of the syntactic (surface form) rewrite rules in Sparser's grammar will have words or semantic categories on their righthand sides resulting from the application of some TAG schema during the definition of a domain-specific type in the world model; when the distinction must be made we will refer to them as 'semantically characterized rules'. There are also rewrite rules whose categories are taken from the conventional syntactic vocabulary and consequently are shared across all of the semantic domains; we call these 'form rules' because they make reference to general aspects of surface form rather ideosyncratic aspects that apply to only certain domains of discourse and their sublanguages. Form rules are the subject of this section. We will look at an example of a form rule and the rationale behind its use, and then go through the 'Check' algorithm that coordinates the selection of semantically characterized or form rules.

A form rule combines a word or occasionally a semantically labeled edge — call it the 'literal' — with the form category of another edge — call that edge the 'head'. The head edge will always lie on the headline of the final tree; the literal will always be a specifier, an auxiliary, or a modifying adjunct or adverb. The purpose of a form rule is to capture a general fact about phrase formation in the language, one whose implications for the semantics of the resulting phrase — its denotation — is the same regardless of the domain.

The implementation of form rules in Sparser allows them to be transparent to the rest of the rules in the grammar. The new edge created by composing the literal and the head edge will have the same category (or some linguistically principled projection of it) as that of its head edge — any semantically characterized rules that might eventually apply to the head edge will apply transparently to the new edge as though the

```
#<form-rule  fpsr1
  :righthand-side  ( #<word ''was''> #<category verb+ed> )
  :new-category :passive
  :syntactic-form  #<category verb-group>
  :referent (:head :right-edge
             :subtype #<past> ) >
```

Figure 6: The applicable form rule for "*was confirmed*"

```
#<edge   e21
  :starts-at #<position 17>
  :ends-at #<position 18>
  :category #<word ''was''>
  :left-daughter #<word ''was''>
  :right-daughter :literal-in-a-rule
  :used-in nil
  :form #<category verbal-auxiliary>
  :referent  nil >
```

Figure 7: The edge that spans the word "*was*"

literal had not been there. The observent reader will already have noticed that the set of rules above was missing the cases needed to cover many likely texts: "... *will be confirmed Wednesday in Seattle*", "... *has been confirmed*", etc. All such combinations — with time and place adjuncts, with complex verbal auxiliaries — are handled uniformly throughout the grammar by form rules.

The exceptions to this uniformity come when there needs to be a domain-specific, semantic (denotational) consequence to a particular combination, as when a verb subcategorizes specifically for a location. In such cases the grammar will simply include a semantically characterized rule for that combination, and the parser will look for that rule in preference to the form rule, as described in section 4.1.

The form rule that applies in the case of "*was confirmed*" is shown in figure 6. (Note that the referent in this rule is not a direct pointer into the model as before but a schematic function. It will take the edge on the right of the two constituents — edge-22 denoting the category 'confirm-in-position — and look in the model for one of its specializations (one of its subtypes) that represents the event having occurred in the past, which will then become the denotation of the new edge.)

Informally speaking, this rule applies in the present example because the word "*was*" is adjacent to edge-22 which has the form category 'verb+ed', and because there is no semantically characterized rule that combines "*was*" and edge-22's primary category, 'confirm-in-position'. The new edge formed by the completion of the rule will have the label 'confirm-in-position+passive', as dictated by the form rule's 'new- category' field in coordination with the schema that defined confirm-in-position and made provisions for that projection from the original category over the verb. (Had there been no new-category field in the rule, then by default the new edge would have taken the same label as the one on the head edge — edge-22.)

Before we can go into the formal algorithm for checking whether edges combine and coordinating the two kinds of rules, we must also show the edge that spans the word "*was*" and discuss its possibly unusual aspects (figure 7).

Note that the label on this edge (its 'category' field) contains a word rather than a category. There are two reasons for this. The first is that allowing words as nonterminals permits an elegant treatment of abbreviations: One can simply rewrite the combination of the abbreviation and its period as the full, unabbreviated word; thereby avoiding any need to redundantly extend grammar to include combination rules for the abbreviation as well as for the full word (e.g. "Company" -> "Co" "."). If the same abbreviation has several expansions then multiple edges will be put into the chart.

The second reason is to simplify the implementation of the parsing algorithm so that it needs to manipulate only a single data type — edges — when looking for rule applications, rather than both edges and words. Any word that is mentioned in a rewrite rule will be covered with an edge when it is scanned, the label on that edge being the word itself. This also permits a simple way to discriminate between known and unknown words in the course of a parse. A known word (one with rules in the grammar) will always have one or more edges over it, if only a trivial edge labeled with that word; unknown words will not have edges, leaving a gap among the preterminal edges of the chart, which can be used to trigger heuristics for dealing with unknown words when they appear in certain contexts.

We should also note that all of Sparser's actions are taken over unary or binary rules. Rules with more than two terms on their righthand sides are converted to a set of rules in Chomsky Normal Form, using a dotted-rule convention to provide names for the intermediate non-terminal labels created in the process. Also, in describing the Check algorithm we will not go into any detail about the larger aspects of the parsing algorithm that determines when or whether two adjacent edges will be checked. Basically the parser operates bottom up and left to right, forming constituents from their heads outwards; however, overlaying that conventional traversal of the space of possible constituents is a moderately complex control structure that reduces the search to a deterministic algorithm, in the sense that every edge formed will be part of the final analysis and every edge will have a unique parent edge in the tree. For some of the particulars see McDonald 1992.

4.1 The algorithm that coordinates checking for both kinds of rules

We can define Sparser's Check operation — its algorithm for seeing whether there is a rule in the grammar that combines two particular adjacent edges — as follows:

Consider two adjacent edges which we will call "Left" and "Right". Left ends at some position p in the chart, and Right begins at this same position p. Left is said to be earlier in the chart — closer to the beginning of the text — because it begins at some position p' (p' being strictly less that p) and because Right ends at some position p" (p" being strictly greater than p). Positions are indexed numerically starting with zero before the first word of the text and increasing in a positive direction, one position between each word. The position at which an edge begins always has a lower numerical index than the position at which it ends; no edge can begin and end at the same position.

Only adjacent edges can be combined to form a new edge. The new edge will start at position p' and end at p". The possibility of two adjacent edges combining is function of their labels, where a label is a word or category in the 'category' field of an edge, or one of a specially designated set of 'form categories' pointed to from the 'form' field of an edge. The form field may be empty.

The Check operation is based on a table compiled from the labels of the rules of the grammar. All rules are binary. The table will associate the righthand sides of those rules, taken as ordered pairs of labels, with the rules themselves. We use the table at runtime by forming an index from the labels of the adjacent constituents and looking the index up in the table to see if that pair of constituents was one of the ones defined by the grammar. If it is, we return the rule located at that index in the table; if it is not, there will be

no rule at that index and we return nil indicating that the edges do not combine. We presently implement the table as a hash on the numerical index; it would lend itself to a hardware implementation as will be clear from its definition.

We form the table as follows. For each rule consider the two labels on its righthand side; call them LL and RL for the first and second (left and right) labels respectively. The rules are taken one at a time in some order: the textual order of their definition in the files of the grammar suffices. We assign each label two unique numbers. One number stands for the label when it appears as the left label of a binary rule (LL); the second stands for when it appears as the right label of a rule (RL). If a label never appears on one of the sides we assign it nil for that component. Numbers for RL labels are assigned starting at 1 and increasing by one with each successive RL up to some 'middle number' choosen to be well above the number of labels ever to appear in the grammar. Numbers for LL labels are assigned starting at that middle number plus one and adding that middle number to the prior LL index for each successive LL. The numbers are then stored with their labels for access during the Check operation.

Entries in the table are formed by adding the numbers for the two righthand side labels of each rule and assigning the rule to the table entry with that numerical index. This sum is guarenteed to be unique by the apt choice of the middle number that divides the two number sequences. It is chosen on the model of adding the two halves of a machine word on a computer. For a 32 bit word length we would select as the middle number 65,536 (two to the sixteenth), giving us a maximum of 64k labels. All of the LL labels will have zeros in their lower sixteen bits; all of the RL labels will have zeros in their upper sixteen bits; combining them with a logical AND will yield a new number unique to that combination of its two halves. Adding the two numbers in software has the same effect.

In checking whether two adjacent edges combine we first check whether their two semantic labels combine; then only if those labels do not combine will we go further and check for the possible combination of one or the other of the semantic labels with the other constituent's form label. To facilitate this, the index assignment algorithm is complicated slightly: In iterating through the grammar's rules we distinguish between regular, semantically characterized rules and form rules. A pair of indexes is stored with each label for each direction. The first number in the pair is used for combinations with a semantic label, the second with a form label. The first numbers are established by iterating through the semantically characterized rules. The second by iterating through the form rules.

Given all this, the Check algorithm proper is the following. Given edges Left and Right as defined above, compute the index in the table of the combination's semantic labels (the 'category' fields of the two edges). For Left, retrieve the right-looking number that was assigned to it — the one it was given for when it appeared as the left of the two labels in a binary rule. Do the opposite for Right. Now add the two numbers, and use the sum as an index into the table. A rule will be returned if the two edges compose, and this rule is used as the basis of the new edge spanning them. If that index into the table is nil, indicating that no semantically characterised rule applies to that edge pair, then go on to check for combinations involving form rules. In the present example there will be no semantically characterised rule for "*was*" plus 'confirm-in-position', so we will move on to check form rules.

There will always be two possible form rules to check for: one combining the category field label of Left and the form field label of Right; the other combining the form label of Left and the category label of Right. Only one of these two pairs will have an index defined for it because of the requirement on form rules that one of the two labels on their righthand sides be either a literal word or a semantic category. We start by computing the table index for the first case, and only if that lookup into the table fails to return a rule do we try the second. If one or the other label fails to have a form option in its stored pair of index numbers then that case is undefined. If neither succeeds, then the two adjacent edges do not compose.

This completes the description of Sparser's Check algorithm. It can be summarized as follows: Its purpose is to determine whether two adjacent edges can be composed to form a new

edge. This is allowed in any of three cases: first if the semantic categories of the two edges match the righthand side of some rule in the grammar; failing that if either the semantic category of the first and the form category of the second correspond to some form rule in the grammar, or visa versa. The test is made by adding numbers that were assigned to the labels as the grammar rules were defined, and using that index as the key into a table that records which pairs correspond to rules.

The prohibition on having a form rule based on a pair of form labels is there to rule out the possibility of ever writing a conventional syntactic grammar in this formalism. We rule this out for two reasons. The first is that allowing "pure" form rules would re-introduce to the grammar the massive ambiguities that syntactic grammars are prone to. The second is that doing so would miss the point of this grammar design, namely that the purpose of a grammar is to facilitate the mapping between form and meaning — between the edges formed over words or phrases and their denotations in the system's world model. We only allow form rules to be used if we are already constructing the projection to some individual in the model, i.e. if the combination is between an edge on the head line which will have a projection and one of its adjuncts (broadly construed).

We have presented the Check algorithm in this much detail because we have seldom seen the rule-lookup algorithms of other parsers described and feel that this kind of information should be added to the literature. In the present case, the very high efficiency of this constant-time lookup operation is especially important because of the very large number of rewrite rules in the grammar. A lookup algorithm that was sensitive to the size of the grammar (or for that matter the presence of any 'order of the size of the grammar' operation within the parser) would strongly penalize the use a semantic grammar. Optimizing algorithms for speed is important even with today's processors because of the ever growing volume of text that must be processed.[2]

5 Defining parsing rules as a side-effect of defining a domain object type

As already discussed, the Sparser language comprehension system works in close consort with a universe of individuals and categories (the individualsU types) that supply the denotations of the words and phrases it parses. This model of the system's world (the topic domains it is competent in) and of the ongoing discourse is based on a set of type definitions, and we are using those definitions as the source of the semantically labeled rewrite rules that Sparser uses. In this section we will look at one such definition and the TAG schema that it uses to create the rules. We begin with a discussion of the schema, shown in figure 8 on page 182.

The 'binding-parameters' and 'labels' fields hold sets of symbols that will be substituted for when the schema is applied to the type-definition: The labels are used to define the rewrite patterns of the individual rewrite rules given in the 'cases' field; the parameters are placeholders for the definition's participant roles. The initial terms in each case (e.g. :subject) are just indicators used by the grammar writer to help organize the cases.

The parenthesized expressions following the indicators are the schematic form of the rules; left and righthand sides are indicated in the obvious way. The :head and :binds indicators after the syntactic part of each rule give its semantic interpretation function. They indicate which of the two edges is the head and what role is filled by the denotation of the other edge.

The label symbols on the lefthand sides of the rules (e.g. S, VP, VG) designate the form labels that are to accompany the corresponding semantic labels that are substitued for them to construct the rules. An example of the final form of one of these rules was given earlier.

[2] One day of the Wall Street Journal averages about half a megabyte. It is available online at about six o'clock in the morning. We will want the results of our information extractions and their pragmatic analysis on the desks of clients when they come in three hours later, and it is only one of any number of online text-based information services that might be used. Sparser processes the sublanguage-rich portions of a text at about 20 words a second (Mac-II, 68020, 15.7mhz); off-topic portions run at about 100 words/sec.

```
(define-exploded-tree-family  transitive/passive
  :binding-parameters ( agent patient )
  :labels ( s vp vg np-subject np-object )
  :cases
     ((:subject (s  (np-subject vp)
                 :head right-edge
                 :binds (agent left-edge))
      (:direct-object (vp  (vg np-object)
                       :head left-edge
                       :binds (patient right-edge)))
      (:passive (s  (np-object vg+passive)
                  :head right-edge
                  :binds (patient left-edge)))
      (:pos-nominalization (s  (np-object+pos vg+nominalization)
                            :head right-edge
                            :binds (patient left-edge)))
      (:of-nominalization (s  (vg+nominalization of+np-object)
                           :head left-edge
                           :binds (patient right-edge)))))
```

Figure 8: TAG schema for rule creation

Space does not permit a full explication of the relationship between this set of independent context-free rule schemas and the full clausal trees of the corresponding TAG tree family (but see McDonald, to appear). Briefly, we independently have an extensive TAG grammar, organized as tree families, that for for many years we have been using in our work on language generation. Here what we have done (by hand at the moment) is take those trees and decompose each of them into their layers of context- free rules, avoiding duplicates since many trees have identical sentential forms at some layer such as S -> NP VP.

The connectedness that characterizes the levels of immediate constituents in a TAG tree is obviously not present in the schema. It is reconstituted when the schema is applied: generic labels like VP that appear in all clauses syntactically and so can provide no connection across the cf rules are replaced during the application with unique semantically characterized labels like confirm-in-position+passive. These create a connection within the instantiated set of cf rules not unlike the quasi-nodes of Vijay Shankar (1993).

Moving on now, consider the definition of the domain-specific model-level type 'confirm-in-position', shown in figure 9 below. The representation language being used is called KRISP, and is described in McDonald (in press); it is a near cousin of Krypton in the Kl-One family, with special features that make it especially well suited as a source representation for language generation, and some efficient data storage and indexing features that suit it to language comprehension.

The first several fields of confirm-in-position are what you would expect in a modern frame language. The category's position in the taxonomic specialization hierarchy is given by its 'specializes' field. Its 'binds' field lists the participants that are specific to this event-type; although an individual instance can have additional participants like time and place by warrant of inheriting from the category TeventU. The participants' value-restrictions are indicated by the :v/r flag. The 'instantiates' field indicates which category above it in the hierarchy an instance should be indexed under in the discourse history; thus for purposes of searching the discourse history, a confirm event is taken to be equivalent to any other job-event, e.g. elect, appoint, resign, etc. The 'index' field specifies how individuals are stored.

```
(define-category confirm-in-position
  :instantiates  job-event
  :specializes   get-position
  :binds ((agent (:v/r :or board-of-directors company))
          (person (:v/r person))
          (position (:v/r position)))
  :index (:temporary :list)
  :realization
    (:tree-family transitive/passive
     :mapping ((agent . agent)
               (patient . person)
               (s . job-event)
               (np-subject . (board-of-directors company))
               (vp . job-event\agent)
               (vg . :main-verb)
               (np/object . person)
     :main-verb ''confirm''
     :saturation (agent person position)
     :additional-rules
        ((:adjunct (job-event  (job-event in-position)
                    :head left-edge
                    :binds (position right-edge)))
         (:adjunct (job-event  (job-event as-position)
                    :head left-edge
                    :binds (position right-edge))) )))
```

Figure 9: The domain-specific model-level type 'confirm-in-position'

The 'realization' field is the basis for instantiating the category's portion of the semantic grammar. The rules constructed from it will recognize phrases denoting instances of individuals with this category in a text, and will guide the semantic interpretation to instantiate them and populate their participant roles. The field indicates which syntactic rule schema should be used, in this case the tree-family 'transitive/passive' shown just above. In its 'mapping' field it indicates the correspondences between the roles of the category and the substitution variables in the schema. The 'main-verb' field (vs. 'head-noun' in other cases) indicates what word should be used as the lexical head of the tree family; its morphological variants are calculated automatically, though if it were irregular the special cases would be given explicitly here. The 'saturation' field indicates that those three roles must be given in a text before an individual of this type is fully defined.

When this form is executed to define the category, its realization field is interpreted and the syntactic schema (tree family) it indicates is applied to the parameters given in the field. Each of the rule cases of the schema is taken up in turn and the corresponding rewrite rule created by making the substitutions called for by the mapping field. Notice that there are two additional cases given in this category's realization field. They identify how the category's 'position' role is filled, namely with either of two different adjunct prepositional phrases, one using "*in*", the other "*as*". (They are analyzed here as attaching to the sentence rather than the verb phrase for compatibility with the way nominalizations are treated.) This kind of local augmentation of the general syntactic schemas (the set of tree families) is convenient for specifying the often ideosyncratic way that individual verbs can subcategorize

for optional complements.

Let us look at one case, the first one in the schema that specifies S -> NP-subject VP. (The NP is annotated as the subject to distinguish it among the substitution variables from the other NP, the direct object.) Looking to the mapping specification we see we have to make two rules because the subject np can take either of two values.We form the lefthand side of the both by substituting the category 'job-event' for S. We form the righthand side of one by substituting the category 'board-of-directors' for the variable 'np-subject' and substituting 'job-event\agent' for 'vp'; the other rule gets 'company' substituted for its np-subject.

The rule must also have a syntactic-form category, which will be the form label of the edge created when the rule completes. It takes this from the syntactic category (substitution variable) given as the rule's lefthand side in the schema (i.e. S). The rule also must have a referent, which it gets from the corresponding part of the case in the schema, subsituting its role 'person' for the symbol 'patient' given in the schema, again as indicated by the mapping field.

We should note in closing this section that a number of largely arbitrary decisions were made in the writing of this grammar, which is to say in the selection of what semantic categories to use in stating the correspondences between semantic labels and syntactic labels in the mapping field of a definition like this. For example we have made a generalization about the semantic interpretations of a whole semantic class of verbs by deciding to specify the VP label as 'job-event\agent' rather than, say, 'confirm-in- position' (which would be as specific as we would be able to be). By choosing the more general category (which should be read as in a categorial grammar: look to compose with an agent to the left of this constituent and label the new edge 'job-event'), we are saying that all the verbs in this family (or rather their model-level denotations) make the same interpretation of their subject NP, i.e. they all map it to their equivalent of the agent role. This points out that linguistic generalizations can be made in the semantic realm as well as the syntactic, and that by using a semantic grammar we are able to express this in a very direct fashion.

6 Concluding remarks

Two substantive complaints have been made against semantic grammars in the literature (e.g. Wallace 1984). One is that they cannot be easily extended as one moves from one domain to the next, while a thorough syntactic grammar can be taken over unchanged. This observation misses a crucial point, however, namely that to understand a text one must find the correspondence between the structural analysis provided by the syntax and the individuals and domain-types in the world model of the application system that is going to use the information given in the text. There is no question that the world model of the first domain that one works with will have to be extended when one moves to the next — the next domain will involve new kinds of objects and new types of relations, and these will have to be added to the model if texts in the new domain are to be understood. There is no other possibility and the fact that there may be no change in the syntactic grammar is besides the point because it is not the whole story.

By using a semantic grammar and having it constructed directly from the world model as done here with Sparser, we will able to keep the grammar in step with the extensions to the model. In particular, we will be continually reminded, as we work on the model, that we need to work out the words and syntactic constructions used to talk about the new concepts, since each domain type should have a realization field to indicate this information.

The second substantive complaint is that semantic grammars are incapable of capturing linguistic generalizations, and so one cannot take advantage of these uniformities when writing the grammar, and one may be tempted to write rules that would be unreasonable from a linguistic perspective and so likely to be brittle in practice (e.g. defining constituents that combine an np with a following preposition, leaving the preposition's complement stranded). This is the more legitimate of the two complaints, but it is also specifically what the present system was designed to address.

By keeping the grammarian (domain modeler) from writing rules directly except in the most ideosyncratic cases (e.g. for dates written as "3/31/93"), and instead forcing them to work by

way of some well-crafted syntactic schemas, we have imposed a discipline on the grammar, forcing it to fit our conception of properly analyzed linguistic structure as captured in the schemas. At the same time we have reduced the overall effort required, since it is markedly more dispatchful to copy and specialize the realization fields of a set of related domain-type definitions than to write out the rules individually.

We have been using semantic grammars in the Sparser system since its inception three years ago. Form rules were introduced after the experiences with its first major grammar, and had a significant effect in simplifying the effort to develop new vocabulary since they allowed whole sets of rules for generic constructions to be used just by giving new semantic categories in the grammar the appropriate syntactic labels. The use of realization fields on model-level definitions and the development of the 'exploded tree families' used as schemas is comparatively recent, and we have only now redone the original grammar for the Who's News domain in its terms. The proof of the pudding, as it were, will come as the models and grammars for the next topic domains are added in the coming months (these will be for joint-ventures and quarterly earnings). If the effort to make these extensions is dramatically smaller than when the original grammars were developed by hand we will judge the design a pragmatic success. If a large family of syntactic schemas, especially for noun phrases, can be developed that addresses the bulk of the construction-types that we see in the business texts we work with then we will have gotten a long way towards solving the problem of extending grammars to new domains.

References

Abeille, A. (1988) *A French Tree Adjoining Grammar*. technical report. Department of Computer & Information Science, University of Pennsylvania.

Burton, R. (1976) *Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems*. Report No. 3453, Bolt Beranek and Newman Inc, Cambridge, MA.

——— — J. S. Brown (1975) "Multiple Representations of Knowledge for Tutorial Reasoning", In: Bobrow & Colins (eds.) *Representation and Understanding*. New York: Academic Press.

Hendrix, G. — E. Sacerdoti — D. Sagalowicz, — J. Slocum (1978) "Developing a natural language interface to complex data" ACM TODS. 3(2), 105 - 147.

Joshi, A..K. (1985) "How much context-sensitivity is required to provide reasonable structural descriptions: tree adjoining grammars". In: Dowty, D.R., L. Kartutunen, A.M. Zwicky (eds) *Natural Language Processing*. Cambridge, U.K.: Cambridge University Press.

Kittredge, R. — J. Lehrberger (1982) *Sublanguage: Studies of Language in Restricted Semantic domains*. Berlin: de Gruter.

McDonald, D. (1992) "An Efficient Chart-based Algorithm for Partial-Parsing of Unrestricted Texts". In: *Proceedings of the 3d Conference on Applied Natural Language Processing* (ACL). Trento, Italy. April 1992. 193 - 200.

——— (to appear) "Reversible NLP by Linking the Grammar to the Knowledge Base". In Strzalkowski, T. (ed), *Reversible Grammar in Natural Language Processing*. Kluwer Academic.

——— (in press) "KRISP: a representation for the semantic interpretation of real texts". *Mind and Machines*.

Riesbeck, C. — C.E. Martin (1985) "Direct Memory Access Parsing". technical report DCS/RR 354. Department of Computer Science, AI Group. Yale University.

Schabes, Y. — A. Abeille — A.K. Joshi (1988) "Parsing Strategies with 'Lexicalized' Grammars: Application to Tree Adjoining Grammars". In: *Proceedings of Coling-88*, Budapest, Hungary.

Vijay-Shanker, K. (1993) "Using Descriptions of Trees in a Tree Adjoining Grammar". *Computational Linguistics*. 18(4), 481 - 517.

Wallace, M. (1984) *Communicating with databases in natural language*. Chichester, U.K.: Ellis Horwood.

Waltz, D. — T. Finin — F. Green — B. Goodman — G. Hadden (1976) *The PLANES system: Natural language access to a large database*. Report T-34. Department of Computer Science, University of Illinois.

Increasing the Applicability of LR Parsing

Mark-Jan Nederhof * Janos J. Sarbo

University of Nijmegen, Department of Computer Science
Toernooiveld, 6525 ED Nijmegen, The Netherlands
E-mail: {markjan,janos}@cs.kun.nl

Abstract

In this paper we describe a phenomenon present in some context-free grammars, called *hidden left recursion*. We show that ordinary LR parsing according to hidden left-recursive grammars is not possible and we indicate a range of solutions to this problem. One of these solutions is a new parsing technique, which is a variant of traditional LR parsing. This new parsing technique can be used both with and without lookahead and the nondeterminism can be realized using backtracking or using a graph-structured stack.

1 Introduction

The class of LR parsing strategies constitutes one of the strongest and most efficient classes of parsing strategies for context-free grammars. LR parsing is commonly used in compilers as well as in systems for the processing of natural language.

Deterministic LR parsing with lookahead of k symbols is possible for LR(k) grammars. Deterministic parsing according to grammars which are not LR(k) can in some cases be achieved with some disambiguating techniques. (Important progress in this field has been reported by Thorup (1992)). However, these techniques are not powerful enough to handle practical grammars for e.g. natural languages.

If we consider LR parsing tables in which an entry may contain multiple actions, then we obtain nondeterministic LR parsing. We will refer to realizations of nondeterministic LR parsing as *generalized* LR parsing. The most straightforward way to obtain generalized LR parsing is by using backtracking (Nilsson, 1986).

A more efficient kind of generalized LR parsing has been proposed by Tomita (1986). The essence of this approach is that multiple parses are processed simultaneously. Where possible, the computation processing two or more parses is shared. This is accomplished by using a *graph-structured stack*.

Although generalized LR parsing can handle a large class of grammars, there is one phenomenon which it cannot handle, viz. *hidden left recursion*. Hidden left recursion, defined at the end of this section, occurs very often in grammars for natural languages.

A solution for handling hidden left-recursive grammars using Tomita's algorithm was proposed by Nozohoor-Farshi (1989). In that paper, the ordinary acyclic graph-structured stack is generalized to allow cycles. The resulting parsing technique is largely equivalent to a parsing technique which follows from a construction defined earlier by Lang (1974), which makes use of a parse matrix. As a consequence, termination of the parsing process is always guaranteed. This means that hidden left-recursive grammars and even cyclic grammars can be handled.

However, cyclic graph-structured stacks may complicate garbage collection and cannot be realized using memo-functions (Leermakers et al., 1992). Tomita's algorithm furthermore becomes very complicated in the case of augmented context-free grammar (e.g. attribute grammar, affix grammar, definite clause gram-

*Supported by the Dutch Organisation for Scientific Research (NWO), under grant 00-62-518

mar, etc.). In this case, different subparses almost always have different attribute values (or affix values, variable instantiations, etc.) and therefore sharing of the computation of context-free parsing would obstruct the correct computation of these values (Nederhof — Sarbo, 1993a).

In this paper we discuss an alternative approach of adapting the (generalized) LR parsing technique to hidden left-recursive grammars.

Our approach can be roughly described as follows. Reductions with epsilon rules are no longer performed. Instead, a reduction with some non-epsilon rule does not have to pop all the members in the right-hand side off the stack; only those which do not derive the empty string must be popped, for others it is optional. The definition of the closure function for sets of items is changed accordingly. Our approach requires the inspection of the parse stack upon reduction in order to avoid incorrect parses.

The structure of this paper is as follows. In the next section we give an introduction to the problem of LR parsing according to hidden left-recursive grammars. We give two naive ways of solving this problem by first transforming the grammar before constructing the (nondeterministic) LR automaton. (These methods are naive because the transformations lead to larger grammars and therefore to much larger LR automata.) We then show how the first of these transformations can be incorporated into the construction of LR automata, which results in parsers with a fewer number of states. We also outline an approach of adapting the LR technique to cyclic grammars.

In Section 3 we prove the correctness of our new parsing technique, called ϵ-LR parsing. Efficient generation of ϵ-LR parsers is discussed in Section 4. We conclude in Section 5 by giving some results on the comparison between the number of states of various LR and ϵ-LR parsers.

We would like to stress beforehand that grammars with nontrivial hidden left recursion can never be handled using deterministic LR parsing (Section 2.5), so that most of the discussion in this paper is not applicable to deterministic LR parsing. We therefore, contrary to custom, use the term "LR parsing" for *generalized* LR parsing, which can at will be realized using backtracking (possibly in combination with memo-functions) or using *acyclic* graph-structured stacks. Where we deal with *deterministic* LR parsing, this is indicated explicitly.

The notation used in the sequel is for the most part standard and is summarized below.

A context-free grammar $G = (T, N, P, S)$ consists of two finite disjoint sets N and T of nonterminals and terminals, respectively, a start symbol $S \in N$, and a finite set of rules P. Every rule has the form $A \rightarrow \alpha$, where the left-hand side (lhs) A is an element from N and the right-hand side (rhs) α is an element from V^*, where V denotes $(N \cup T)$. P can also be seen as a relation on $N \times V^*$.

We use symbols $A, B, C, \ldots$ to range over N, symbols X, Y, Z to range over V, symbols $\alpha, \beta, \gamma, \ldots$ to range over V^*, and $v, w, x, \ldots$ to range over T^*. We let ϵ denote the empty string. A rule of the form $A \rightarrow \epsilon$ is called an *epsilon rule*.

The relation P is extended to a relation $\overset{G}{\rightarrow}$ on $V^* \times V^*$ as usual. We write $\rightarrow$ for $\overset{G}{\rightarrow}$ when G is obvious. The transitive closure of $\rightarrow$ is denoted by $\rightarrow^+$ and the reflexive and transitive closure is denoted by $\rightarrow^*$.

We define: $B \lessdot A$ if and only if $A \rightarrow B\alpha$ for some α. The transitive closure of $\lessdot$ is denoted by $\lessdot^+$.

We distinguish between two cases of left recursion. The most simple case, which we call *plain left recursion*, occurs if there is a nonterminal A such that $A \lessdot^+ A$. The other case, which we call *hidden left recursion*, occurs if $A \rightarrow B\alpha$, $B \rightarrow^* \epsilon$, and $\alpha \rightarrow^* A\beta$, for some A, B, α, and β; the left recursion is "hidden" by the empty-generating nonterminal. (An equivalent definition of hidden left recursion is due to Leermakers et al. (1992).)

A grammar is said to be *cyclic* if $A \rightarrow^+ A$ for some nonterminal A.

A nonterminal A is said to be *nonfalse* if $A \rightarrow^* \epsilon$. A nonterminal A is called a *predicate* if it is nonfalse and $A \rightarrow^* v$ only for $v = \epsilon$.[1]

[1]The terms "nonfalse" and "predicate" seem to

We call a nonterminal A *reachable* if $S \rightarrow^* \alpha A \beta$ for some α and β. We call a grammar *reduced* if every nonterminal is reachable and derives some terminal string. Where we give a transformation between context-free grammars, we tacitly assume that the input grammars are reduced and for these grammars the output grammars are guaranteed also to be reduced.

2 Hidden left recursion and LR parsing

The simplest nontrivial case of hidden left recursion is the grammar G_1 given by the following rules.

$$\begin{array}{lcl} A & \rightarrow & BAc \\ A & \rightarrow & a \\ B & \rightarrow & b \\ B & \rightarrow & \epsilon \end{array}$$

In this grammar, nonterminal A is left-recursive. This fact is hidden by the presence of a nonfalse nonterminal B in the rule $A \rightarrow BAc$. Note that this grammar is ambiguous, as illustrated in Figure 1. This is typically so in the case where the one or more nonfalse nonterminals which hide the left recursion are not all predicates.

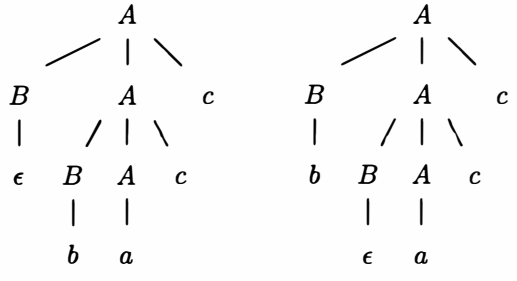


Figure 1: Two parse trees with the same yield, showing ambiguity of G_1.

have been used for the first time by Knuth (1971) and Koster (1971), respectively, although in a slightly different meaning.

2.1 Generalized LR parsing and hidden left recursion

We now discuss informally how (generalized) LR parsing fails to terminate for the above grammar. We assume that the reader is familiar with the construction of (nondeterministic) LR(0) automata. Our terminology is taken from Aho et al. (1986).

A pictorial representation of the LR(0) parsing table for G_1 is given in Figure 2. LR parsing of any input w may result in many sequences of parsing steps, one of which is illustrated by the following sequence of configurations.

Stack contents	Inp.	Action
Q_0	w	red($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1$	w	red($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1\ B\ Q_1$	w	red($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1\ B\ Q_1\ B\ Q_1$	w	red($B \rightarrow \epsilon$)
⋮	⋮	⋮

The sequence of parsing steps illustrated above does not terminate. We can find a non-terminating sequence of parsing steps for the LR(0) automaton for every hidden left-recursive grammar. In fact, this is even so for the LR(k), LALR(k), and SLR(k) automata, for any k. Hidden left recursion has been identified by Soisalon-Soininen — Tarhio (1988) as one of two sources, together with cyclicity, of the looping of LR parsers.

Various other parsing techniques, such as left-corner parsing (Nederhof, 1993a) and cancellation parsing (Nederhof, 1993b), also suffer from this deficiency.

2.2 Eliminating epsilon rules

We first discuss a method to allow LR parsing for hidden left-recursive grammars by simply performing a source to source transformation on grammars to eliminate the rules of which the right-hand sides only derive the empty string. To preserve the language, for each rule containing an occurrence of a nonfalse nonterminal a copy must be added without that occurrence. Following Aho — Ullman (1972), this transformation, called ϵ-*elim*, is described below. The input grammar is called G.

1. Let G_0 be G.

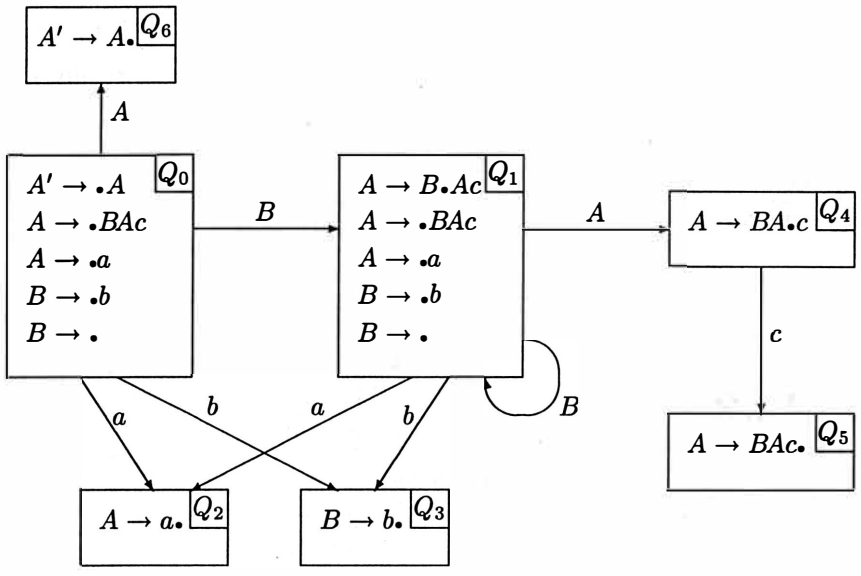


Figure 2: The LR(0) automaton for G_1.

2. Remove from G_0 all rules defining predicates in G and remove all occurrences of these predicates from the rules in G_0.

3. Replace every rule of the form $A \to \alpha_0 B_1 \alpha_1 B_2 \dots B_m \alpha_m$ in G_0, $m \geq 0$, where the members which are nonfalse in G are exactly $B_1, \dots, B_m$, by the set of rules of the form $A \to \alpha_0 \beta_1 \alpha_1 \beta_2 \dots \beta_m \alpha_m$, where β_i is either B_i or ϵ and $\alpha_0 \beta_1 \alpha_1 \beta_2 \dots \beta_m \alpha_m \neq \epsilon$. Note that this set of rules is empty if $m = 0$ and $\alpha_0 = \epsilon$, in which case the original rule is just eliminated from G_0.

4. If S is nonfalse in G, then add the rules $S^\dagger \to S$ and $S^\dagger \to \epsilon$ to G_0 and make $S^\dagger$ the new start symbol of G_0. (In the pathological case that S is a predicate in G, $S^\dagger \to S$ should of course not be added to G_0.)

5. Let ϵ-*elim*(G) be G_0.

Note that for every rule $A \to \alpha$ such that α contains k occurrences of nonfalse non-predicates, the transformed grammar may contain 2^k rules.

In this paper, an expression of the form $[B]$ in a rhs indicates that the member B has been eliminated by the transformation. It is for reasons of clarity that we write this expression instead of just leaving B out.

An item of the form $A \to [\alpha_0] X_1 [\alpha_1] \dots [\alpha_{i-1}] . X_i \dots X_m [\alpha_m]$ is said to be *derived* from the *basic* item $A \to \alpha_0 X_1 \alpha_1 \dots \alpha_{i-1} . X_i \dots X_m \alpha_m$.[2] According to the convention mentioned above, $A \to \alpha_0 X_1 \alpha_1 \dots X_m \alpha_m$ is a rule in G, and $A \to X_1 \dots X_m$ is a rule in ϵ-*elim*(G). The item of the form $S^\dagger \to .$ which may be introduced by ϵ-*elim* will be regarded as the derived item $S^\dagger \to [S].$.

Example 2.1 Let the grammar G_2 be defined by the rules

$$\begin{array}{lcl} A & \to & BCD \\ B & \to & \epsilon \\ B & \to & b \\ C & \to & \epsilon \\ D & \to & \epsilon \\ D & \to & d \end{array}$$

Step 2 of ϵ-*elim* removes the rule $C \to \epsilon$ defining the only predicate C. Also the occur-

[2] We avoid writing dots in dotted items immediately to the left of eliminated members.

rence of C in $A \rightarrow BCD$ is removed, i.e. this rule is replaced by $A \rightarrow B[C]D$.

Step 3 removes all rules with an empty rhs, viz. $B \rightarrow \epsilon$ and $D \rightarrow \epsilon$, and replaces $A \rightarrow B[C]D$ by the set of all rules which result from either eliminating or retaining the nonfalse members, viz. B and D (C is not a member anymore!), such that the rhs of the resulting rule is not empty. This yields the set of rules

$$\begin{array}{lcl} A & \rightarrow & B[C]D \\ A & \rightarrow & B[CD] \\ A & \rightarrow & [BC]D \end{array}$$

Step 4 adds the rules $A^\dagger \rightarrow A$ and $A^\dagger \rightarrow \epsilon$. The new start symbol is $A^\dagger$.

We have now obtained ϵ-*elim*(G_2), which is defined by

$$\begin{array}{lcl} A^\dagger & \rightarrow & A \\ A^\dagger & \rightarrow & \epsilon \\ A & \rightarrow & B[C]D \\ A & \rightarrow & B[CD] \\ A & \rightarrow & [BC]D \\ B & \rightarrow & b \\ D & \rightarrow & d \end{array}$$ □

Note that in the case that ϵ-*elim* introduces a new start symbol $S^\dagger$, there is no need to augment the grammar (i.e. add the rule $S' \rightarrow S^\dagger$ and make S' the new start symbol) for the purpose of constructing the LR automaton. Augmentation is in this case superfluous because the start symbol $S^\dagger$ is not recursive.

In the case of G_1, the transformation yields the following grammar.

$$\begin{array}{lcl} A & \rightarrow & BAc \\ A & \rightarrow & [B]Ac \\ A & \rightarrow & a \\ B & \rightarrow & b \end{array}$$

The LR(0) table for this grammar is represented in Figure 3.

Together with the growing number of rules, the above transformation may also give rise to a growing number of states in the LR(0) automaton. In the above case, the number of states increases from 7 to 8, as indicated by Figures 2 and 3. As G_1 is only a trivial grammar, we may expect that the increase of the number of states for practical grammars is much larger. Tangible results are discussed in Section 5.

2.3 A new parsing algorithm

To reduce the number of states needed for an LR automaton for ϵ-*elim*(G), we incorporate the transformation in the closure function. This requires changing the behaviour of the LR automaton upon reduction.

This approach can in a different way be explained as follows. Items derived from the same basic item by ϵ-*elim* are considered the same. For instance, the items $A \rightarrow BAc\bullet$ and $A \rightarrow [B]Ac\bullet$ in Figure 3 are considered the same because they are derived from the same basic item $A \rightarrow BAc\bullet$.

All items are now represented by the basic item from which they are derived. For instance, both items in Q_5 in Figure 3 are henceforth represented by the single basic item $A \rightarrow BAc\bullet$. The item $A \rightarrow [B]Ac\bullet$ in state Q_7 is now represented by $A \rightarrow BAc\bullet$.

As a result, some pairs of states now consist of identical sets of items and may therefore be merged. For the example in Figure 3, the new collection of states is given in Figure 4. It can be seen that states Q_5 and Q_7 are merged into state $Q_{5/7}$.

In the resulting LR table, it is no longer indicated which derived items are actually represented. Correspondingly, the behaviour of the new automaton is such that upon reduction all possibilities of derived items are nondeterministically tried.

For instance, consider the parsing of *bacc* using the LR(0) automaton in Figure 4. The first sequence of parsing steps is without complications:

Stack contents	Inp.	Action
Q_0	*bacc*	shift
$Q_0\ b\ Q_3$	*acc*	red($B \rightarrow b$)
$Q_0\ B\ Q_1$	*acc*	shift
$Q_0\ B\ Q_1\ a\ Q_2$	*cc*	red($A \rightarrow a$)
$Q_0\ B\ Q_1\ A\ Q_4$	*cc*	shift
$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	*c*	red(?)

Now there are two ways to perform a reduction with the item $A \rightarrow BAc\bullet$. One way is to pretend that B has been eliminated from this rule. In other words, we are dealing with the derived item $A \rightarrow [B]Ac\bullet$. In this case we remove two states and grammar symbols from the stack. The sequence of configurations from here on now begins with

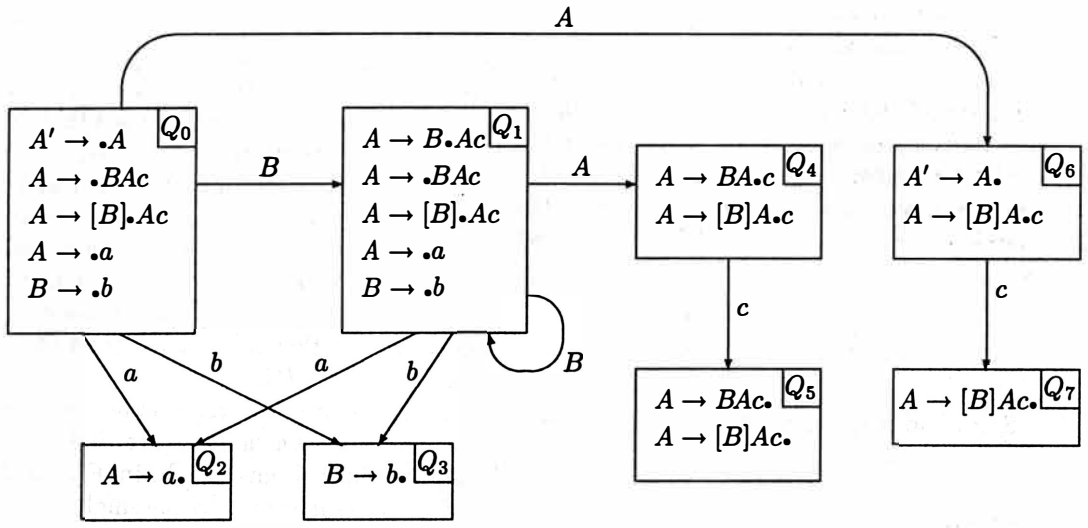


Figure 3: The LR(0) automaton for ϵ-$elim(G_1)$.

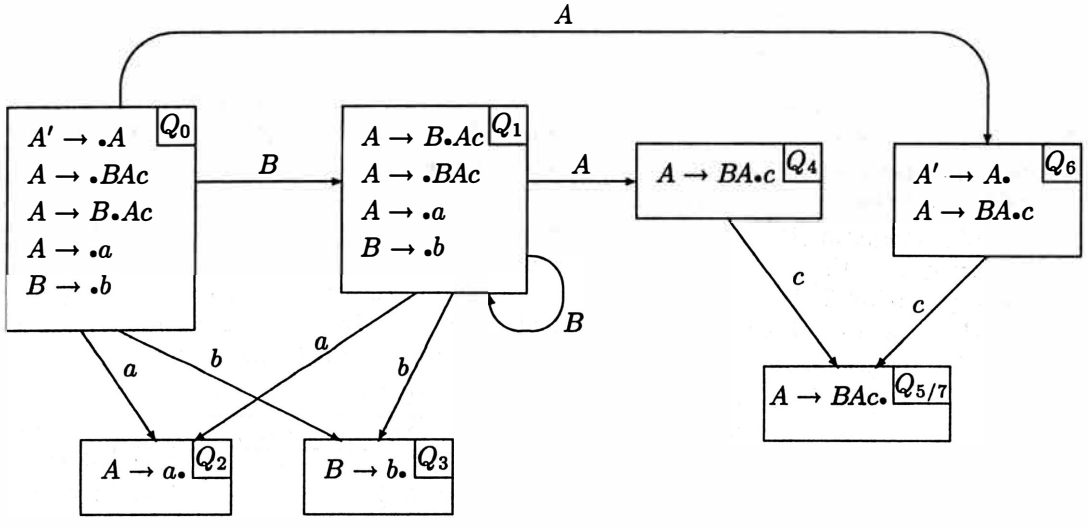


Figure 4: The optimised LR(0) automaton for ϵ-$elim(G_1)$ with merged states.

$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	c	red($A \to [B]Ac$)
$Q_0\ B\ Q_1\ A\ Q_4$	c	
⋮	⋮	⋮

The other way to perform reduction is by taking off the stack all the members in the rule $A \to BAc$ and the same number of states, and we obtain

$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	c	red($A \to BAc$)
$Q_0\ A\ Q_6$	c	shift
$Q_0\ A\ Q_6\ c\ Q_{5/7}$		red(?)

We are now at an interesting configuration. We have again the choice between reducing

with $A \rightarrow [B]Ac$ or with the unaffected rule $A \rightarrow BAc$. However, it can be established that reduction with $A \rightarrow BAc$ is not possible, because there is no B on the stack to be popped.

At this point, the main difference between traditional LR parsing and the new parsing technique we are developing becomes clear. Whereas for traditional LR parsing, the grammar symbols on the stack have no other purpose except an educational one, for our new parsing technique, the investigation of the grammar symbols on the stack is essential for guiding correct parsing steps.

In general, what happens upon reduction is this. Suppose the state on top of the stack contains an item of the form $A \rightarrow \alpha\bullet$, then reduction with this item is performed in the following steps.

1. The parser nondeterministically looks for some sequence of grammar symbols X_1, ..., X_m such that there are α_0, ..., α_m with
   - $\alpha = \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$
   - $\alpha_0 \rightarrow^* \epsilon \wedge \ldots \wedge \alpha_m \rightarrow^* \epsilon$
   - The top-most m grammar symbols on the stack are $X_1, \ldots, X_m$ in that order, i.e. X_1 is deepest in the stack and X_m is on top of the stack.
   - $m = 0 \Rightarrow A = S'$

   In words, α is divided into a part which is on the stack and a part which consists only of nonfalse nonterminals. The part on the stack should not be empty with the exception of the case where $A \rightarrow \alpha$ is the rule $S' \rightarrow S$.
2. The top-most m symbols and states are popped off the stack.
3. Suppose that the state on top of the stack is Q, then
   - if $A = S'$, then the input is accepted, provided Q is the initial state and the end of the input has been reached; and
   - if $A \neq S'$, then A and subsequently *goto* (Q, A) are pushed onto the stack, provided *goto* (Q, A) is defined (otherwise this step fails).

The way the reduction is handled above corresponds with the reduction with the rule $A \rightarrow [\alpha_0]X_1[\alpha_1]\ldots X_m[\alpha_m]$ in the original LR(0) parser for ϵ-*elim*(G).

Incorporating the transformation ϵ-*elim* into the construction of the LR table can be seen as a restatement of the usual closure function, as follows.

$$
\begin{aligned}
&\mathit{closure}\ (q) = \\
&\quad \{B \rightarrow \delta\bullet\theta \mid A \rightarrow \alpha\bullet\beta \in q \wedge \beta \rightarrow^* B\gamma \wedge \\
&\qquad\qquad B \rightarrow \delta\theta \wedge \\
&\qquad\qquad \exists v[v \neq \epsilon \wedge \delta\theta \rightarrow^* v] \wedge \\
&\qquad\qquad \delta \rightarrow^* \epsilon\} \\
&\quad \cup \\
&\quad \{A \rightarrow \alpha\delta\bullet\beta \mid A \rightarrow \alpha\bullet\delta\beta \in q \wedge \delta \rightarrow^* \epsilon\}
\end{aligned}
$$

Note that the expression $\beta \rightarrow^* B\gamma$ allows nonterminals to be rewritten to the empty string. Also note that $\exists v[v \neq \epsilon \wedge \delta\theta \rightarrow^* v]$ excludes rules of which the rhs can only derive ϵ. Efficient calculation of the closure function is investigated in Section 4.1.

Leermakers (1992) proposes similar changes to some functions in the recursive ascent Earley parser in order to allow hidden left recursion. Similar changes were made by Graham et al. (1980) in order to improve the efficiency of Earley parsing. We have recently learned that a parsing technique very similar to ours is suggested by Leermakers (1993).

The investigation of the grammar symbols on the stack for the purpose of guiding correct parsing steps is reminiscent of Pager (1970), who proposed a general method for the compression of parsing tables by means of merging states. If the full stack may be investigated upon reduction, then the need for states in the traditional sense is even completely eradicated, as shown by Fortes Gálves (1992).[3]

In Section 3 we prove the correctness of the new parsing technique, which we call ϵ-LR parsing.

[3] It is interesting to note that various degrees of simplification of the collection of sets of items are possible. For example, one could imagine an approach half-way between our approach and the one by Fortes, according to which items consist only of the parts which occur normally after the dots. This leads to even more merging of states but requires more effort upon reductions.

2.4 Dealing with cyclic grammars

If needed, ϵ-LR parsing can be further refined to handle cyclic grammars.

The starting-point is again a transformation on grammars, called *C-elim*, which eliminates all *unit rules*, i.e. all rules of the form $A \to B$. This transformation consists of the following steps.

1. Let G_0 be G.
2. Replace every non-unit rule $A \to \alpha$ in G_0 by the set of rules of the form $B \to \alpha$ such that $B \overset{G}{\to}{}^* A$ and either $B = S$ or B has an occurrence in the rhs of some non-unit rule.
3. Remove all unit rules from G_0.
4. Let *C-elim*(G) be G_0.

Termination of LR parsing according to *C-elim*(ϵ-*elim*(G)) is guaranteed for any G.

If we incorporate *C-elim* into the behaviour of our ϵ-LR parsers, then reduction with $A \to \alpha$ is performed by the following steps.

1. The parser nondeterministically looks for some sequence of grammar symbols $X_1, \ldots, X_m$ such that there are $\alpha_0, \ldots, \alpha_m$ with
   - $\alpha = \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$
   - $\alpha_0 \to^* \epsilon \wedge \ldots \wedge \alpha_m \to^* \epsilon$
   - The top-most m grammar symbols on the stack are $X_1, \ldots, X_m$.
   - $m = 0 \Rightarrow A = S'$
   - $m = 1 \Rightarrow (X_1 \in T \vee A = S')$
2. The top-most m symbols and states are popped off the stack.
3. Suppose that the state on top of the stack is Q, then
   - if $A = S'$, then the input is accepted, provided Q is the initial state and the end of the input has been reached; and
   - if $A \neq S'$, then the parser nondeterministically looks for some nonterminal B such that $B \to^* A$ and *goto* (Q, B) is defined, and then B and subsequently *goto* (Q, B) are pushed onto the stack.

Note that the parser which performs reduction in this way, using the parse tables from the ϵ-LR parser, may go into unnecessary dead alleys of length one. This may be avoided by reformulating the closure function such that rules containing a single non-predicate in their right-hand sides are left out.

How to avoid reductions with unit rules (*unit reductions*) in the case of deterministic LR parsing has been investigated in a number of papers (e.g., Heilbrunner, 1985). Our particular technique of avoiding unit reductions is reminiscent of an optimization of Earley's algorithm (Graham et al., 1980).

In the remaining part of this paper, the term "ϵ-LR parsing" will not include the extra extension to ϵ-LR parsing described in this section.

2.5 Applicability of ϵ-LR parsing

In the same way as generalized LR(0) parsing can be refined to generalized SLR(k), LALR(k), and LR(k) parsing ($k > 0$) we can also refine ϵ-LR(0) parsing to ϵ-SLR(k), ϵ-LALR(k), and ϵ-LR(k) parsing. The construction of ϵ-LR tables for these parsing strategies can be adopted from the construction of their LR counterparts in a reasonably straightforward way.

We have shown that ϵ-LR parsing can be used for hidden left-recursive grammars, which cannot be handled using ordinary LR parsing. The variants of ϵ-LR parsing which apply lookahead are useful for making the parsing process more deterministic, i.e. to reduce the number of entries in the parsing table that contain multiple actions.

However, adding lookahead cannot yield completely deterministic parsers in the case of hidden left recursion where at least one of the hiding nonterminals is not a predicate. This is because such a grammar is ambiguous, as discussed earlier. (If all hiding nonterminals are

predicates, then we are dealing with a trivial form of hidden left recursion, which can easily be eliminated by eliminating the hiding nonterminals.)

Also in the case of grammars without hidden left recursion, ϵ-LR parsing may have an advantage over ordinary (generalized) LR parsing: the parsing actions corresponding with subtrees of the parse tree which have empty yields are avoided. For these grammars, the application of lookahead may serve to construct deterministic ϵ-LR parsers.

Nederhof (1993a) describes how subtrees which have empty yields can be attached to the complete parse tree without actually parsing the empty string.

2.6 Specific elimination of hidden left recursion

For the sake of completeness, we describe a way of getting rid of hidden left recursion without using epsilon rule elimination. The idea is that we selectively remove occurrences of nonfalse nonterminals which hide left recursion. In case of a nonfalse non-predicate A, we replace the occurrence of A by an occurrence of a new nonterminal A'. This A' is constructed so as to derive the same set of strings as A does, with the exception of ϵ.

The transformation, constructing grammar *HLR-elim*(G) from grammar G, consists of the following steps.

1. Let G_0 be G.

2. For every rule $A \rightarrow B\alpha$ in G_0 which leads to a hidden left-recursive call (i.e. $\alpha \stackrel{G}{\rightarrow}{}^* A\beta$ for some β, and $B \stackrel{G}{\rightarrow}{}^* \epsilon$), replace the rule by $A \rightarrow \alpha$, and also add $A \rightarrow B'\alpha$ to G_0 provided B is not a predicate in G. Repeat this step until it can no longer be applied.

3. For every new nonterminal A' introduced in G_0 in step 2, or by an earlier iteration of step 3, and for every rule $A \rightarrow \alpha$ in G_0, add to G_0 the rule

   - $A' \rightarrow \alpha$ if not $\alpha \stackrel{G}{\rightarrow}{}^* \epsilon$, or rules of the form
   - $A' \rightarrow X_i' X_{i+1} \ldots X_n$ if $\alpha \stackrel{G}{\rightarrow}{}^* \epsilon$, where $\alpha = X_1 \ldots X_n$, and X_i is not a predicate.

4. Remove from G_0 all rules $A \rightarrow \alpha$ such that A was rendered unreachable by the elimination of rules in step 2.

5. Let *HLR-elim*(G) be G_0.

Example 2.2 Let the grammar G_3 be defined by

$$\begin{array}{lcl} A & \rightarrow & ABAa \\ A & \rightarrow & AAB \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

The grammar *HLR-elim*(G_3) is given by

$$\begin{array}{lcl} A & \rightarrow & Aa \\ A & \rightarrow & A'BAa \\ A & \rightarrow & AB \\ A & \rightarrow & A'AB \\ A & \rightarrow & \epsilon \\ A' & \rightarrow & Aa \\ A' & \rightarrow & A'BAa \\ A' & \rightarrow & A'B \\ A' & \rightarrow & A'AB \\ B & \rightarrow & \epsilon \end{array}$$

□

The transformation *HLR-elim* is very often incorporated in the construction of parsers which can deal with hidden left recursion. An example is the variant of backtrack left-corner parsing as applied in Programmar (Meijer, 1986). See also Nederhof (1993a).

The size of the grammar resulting from the application of this transformation is much smaller than that in the case of *ϵ-elim*. In fact it is only quadratic in the size of the original grammar.

3 Correctness of ϵ-LR parsing

A formal derivation of ϵ-LR(0) parsing is given by Nederhof — Sarbo (1993b). In this section we prove the correctness of ϵ-LR parsing by assuming the correctness of (nondeterministic) LR parsing, which has already been established in literature.

In Section 2.3 we derived the new parsing technique of ϵ-LR parsing. We showed that this kind of parsing is based on traditional LR parsing, with the following differences:

- Instead of using the original grammar G, the transformed grammar ϵ-*elim*(G) is used.
- No distinction is made between items derived from the same basic item. This can be seen as merging states of the LR automaton of ϵ-*elim*(G).
- Because considering derived items as the same leads to a loss of information, a new mechanisms is introduced, which checks upon reduction whether the members of the applied rule are actually on the stack and whether the goto function is defined for the lhs and the state which is on top of the stack after the members are popped.

Because the transformation ϵ-*elim* preserves the language and because we assume the correctness of LR parsing, the correctness of ϵ-LR parsing can be proved by mentioning two points:

- The symbols on the stack and the remaining input together derive the original input, which can be proved by induction on the length of a sequence of parsing steps. This argument shows that no incorrect derivations can be found.
- For every sequence of parsing steps performed by an LR parser (LR(k), SLR(k), etc.) for ϵ-*elim*(G) there is a corresponding sequence of parsing steps performed by the corresponding type of ϵ-LR parser (ϵ-LR(k), ϵ-SLR(k), etc.) for G.

  This proves that ϵ-LR parsing cannot fail to find correct derivations by the assumption that LR parsing according to ϵ-*elim*(G) does not fail to find correct derivations.

In case of ϵ-LR(0) and ϵ-SLR parsing it can also be shown that the set of sequences of parsing steps is isomorphic with the set of sequences of the LR(0) or SLR parsers for ϵ-*elim*(G), and that the corresponding sequences are equally long. It is sufficient to prove that if a reduction can be successfully performed in an ϵ-LR parser, then it can be performed in an LR parser in the corresponding configuration.

For this purpose, suppose that in an ϵ-LR parser some reduction is possible with the item $A \to \alpha_0 A_1 \alpha_1 \ldots A_m \alpha_m \bullet \in Q_m$ such that

- $\alpha_i \to^* \epsilon$ for $0 \leq i \leq m$,
- the topmost $2m+1$ elements of the stack are $Q_0 A_1 Q_1 \ldots A_m Q_m$,
- the goto function for Q_0 and A is defined,
- in the corresponding configuration in the LR parser, the states corresponding with Q_i are called Q'_i.

From the fact that the goto function is defined for Q_0 and A we know that it is also defined for Q'_0 and A and that the item $A \to [\alpha_0]\bullet A_1[\alpha_1]\ldots A_m[\alpha_m]$ is in Q'_0. This implies that $A \to [\alpha_0]A_1[\alpha_1]\ldots A_i[\alpha_i]\bullet\ldots A_m[\alpha_m]$ is in Q'_i because Q'_i is *goto* (Q'_{i-1}, A_i), for $1 \leq i \leq m$.

Therefore, in the corresponding LR parser a reduction would also take place according to the item $A \to [\alpha_0]A_1[\alpha_1]\ldots A_m[\alpha_m]\bullet$.

Regrettably, an isomorphism between sequences of parsing steps of ϵ-LR parsers and the corresponding LR parsers is not possible for ϵ-LR(k) and ϵ-LALR(k) parsing, where $k > 0$. This is because merging derived items causes loss of information on the lookahead of items. This causes the parser to be sent up blind alleys which are not considered by the corresponding LR parser.

Because ϵ-LR parsing retains the prefix-correctness of traditional LR parsing (that is, upon incorrect input the parser does not move its input pointer across the first invalid symbol), the blind alleys considered by an ϵ-LR parser but not the corresponding LR parser are of limited length, and therefore unimportant in practical cases.

Theoretically however, the extra blind alleys may be avoided by attaching the lookahead information not to the state on top of the stack before reduction but to the state on top after popping m states and grammar symbols off the stack (m as in Section 2.3). This means that we have lookahead (a set of

strings, each of which not longer than k symbols) for each state q and nonterminal A such that *goto* (q, A) is defined.

In the cases we have examined, the number of pairs (q, A) for which *goto* (q, A) is defined is larger than the total number of items $A \to \alpha\bullet$ in all states (about 4 to 25 times as large), so this idea is not beneficial to the memory requirements of storing lookahead information. In the case of ϵ-LR(k) parsing ($k > 0$), this idea may however lead to a small reduction of the number of states, since some states may become identical after the lookahead information has been moved to other states.

4 Calculation of items

In this section we investigate the special properties of the closure function for ϵ-LR parsing. First we discuss the closure function for ϵ-LR(k) parsing and then the equivalent notion of kernel items in ϵ-LR parsing.

4.1 The closure function for ϵ-LR(k) parsing

If w is a string and k a natural number, then $k : w$ denotes w if the length of w is less than k, and otherwise it denotes the prefix of w of length k. We use lookaheads which may be less than k symbols long to indicate that the end of the string has been reached.

The initial state for ϵ-LR(k) parsing ($k > 0$) is

$$Q_0 = \mathit{closure}\ (\{[S' \to \bullet S, \epsilon]\})$$

The closure function for ϵ-LR(k) parsing is

$$\begin{aligned}
&\mathit{closure}\ (q) = \\
&\quad \{[B \to \delta\bullet\theta, x] \mid \\
&\qquad [A \to \alpha\bullet\beta, w] \in q \wedge \beta \to^* B\gamma \wedge \\
&\qquad B \to \delta\theta \wedge \\
&\qquad \exists v[v \neq \epsilon \wedge \delta\theta \to^* v] \wedge \\
&\qquad \delta \to^* \epsilon \wedge \\
&\qquad \exists y[\gamma \to^* y \wedge x = k : yw]\} \\
&\quad \cup \\
&\quad \{[A \to \alpha\delta\bullet\beta, w] \mid \\
&\qquad [A \to \alpha\bullet\delta\beta, w] \in q \wedge \delta \to^* \epsilon\}
\end{aligned}$$

4.2 The determination of smallest representative sets

In traditional LR parsing, items are divided into *kernel* items and *nonkernel* items. Kernel items are $S' \to \bullet S$ and all items whose dots are not at the left end. The nonkernel items are all the others. (At this stage we abstain from lookahead.)

As we will only be looking in this section at sets of items which are either Q_0 or of the form *goto* (q, X), which result after application of the closure function, we have that the kernel items from a set of items q are a *representative subset* of q. This means that we can

- construct the complete set of items q by applying the closure function to the representative subset, and
- determine whether two sets of items are equal by determining the equality of their representative subsets.

Because the set of kernel items from a set q is in general much smaller than q itself, kernel items are very useful for the efficient generation of LR parsers.

Regrettably, in the case that the grammar contains many epsilon rules, the set of kernel items from a set q may not be much smaller than q itself. Therefore, kernel items are not very useful for generation of ϵ-LR parsers.

Another approach to finding representative subsets for traditional LR parsing can be given in terms of the stages in which the goto function is executed. According to this principle, the representative subset of *goto* (q, X) is

$$K(q, X) = \{A \to \alpha X\bullet\beta | A \to \alpha\bullet X\beta \in q\}$$

and other items in *goto* (q, X) are obtained by applying the closure function to $K(q, X)$.

In the case of traditional LR parsing, K computes exactly the kernel items in *goto* (q, X), and therefore the two methods for finding representative subsets are equivalent. That this does not hold for ϵ-LR parsing can be easily seen by investigating the definition of *closure* in Section 2.3: according to the second part

$$\{A \to \alpha\delta\bullet\beta \mid A \to \alpha\bullet\delta\beta \in q \wedge \delta \to^* \epsilon\}$$

in this definition, the dot can be shifted over nonfalse members and therefore new items can be added whose dots are not at the left end. Therefore, some kernel items may not be in $K(q, X)$.

It turns out that we can also not use K for finding representative subsets in the case of ϵ-LR parsing. The reason is that K does not provide a *well-defined* method to find representative subsets. I.e. for some grammars we can find sets of items q_1 and q_2 and symbols X and Y such that $goto\ (q_1, X) = goto\ (q_2, Y)$ but $K(q_1, X) \neq K(q_2, Y)$.

The solution that we propose is more refined than the methods in traditional LR parsing.

First, we determine the equivalence relation of mutually left-recursive nonterminals, whose classes are denoted by $[A]$. Thus, $[A] = \{B | A \rightarrow^* B\alpha \wedge B \rightarrow^* A\beta\}$.

A nice property of these classes is that $A \rightarrow \bullet\alpha \in q$ and $B \in [A]$ together imply that $B \rightarrow \bullet\beta \in q$ for every rule $B \rightarrow \beta$. Using this fact, we can replace every item $A \rightarrow \bullet\alpha$ in q by $[A]$ without loss of information.

We define the set Z to be the union of the set of all items and the set of equivalence classes of mutually left-recursive nonterminals. The variables $E, E', \ldots$ range over elements from Z.

Our goal is to find a representative set $q' \subseteq Z$ for each set of items q.

First, we define the binary relation *induces* on elements from Z such that

- *induces* (I, J) for items I and J if and only if $I = A \rightarrow \alpha\bullet B\beta$ and $J = A \rightarrow \alpha B\bullet\beta$ and $B \rightarrow^* \epsilon$
- *induces* (I, E) for item I and class E if and only if $I = A \rightarrow \alpha\bullet B\beta$ and $B \in E$
- *induces* (E, E') for classes E and E' if and only if $E \neq E'$ and there are $A \in E$ and $B \in E'$ such that $A \rightarrow \alpha B\beta$ and $\alpha \rightarrow^* \epsilon$
- *induces* (E, I) for class E and item I if and only if there is $A \in E$ such that $I = A \rightarrow \alpha\bullet\beta$ and $\alpha \rightarrow^* \epsilon$

The smallest set $repr\ (q) \subseteq Z$ representing a set of items q can now be determined by the following steps:

1. Determine $q_1 \subseteq Z$ by replacing in q every item $A \rightarrow \bullet\alpha$ by $[A]$.
2. Let q_2 be the subset of q_1 which results from eliminating all items I such that *induces* (E, I) for some equivalence class $E \in q_1$.
3. Determine the set $repr\ (q)$ defined by $\{E \in q_2 | \neg\exists E' \in q_2 [induces\ (E', E)\}$.

The reason that no information is lost in step 3 is that the relation *induces* restricted to q_2 is not cyclic.

That $repr\ (q)$ is the smallest set $q' \subseteq Z$ representing q can be formalized by stating that it is the smallest subset q' of Z such that $closure\ (q') = q$, where the definition of *closure* is redefined to

$$
\begin{aligned}
&closure\ (q) = \\
&\quad \{B \rightarrow \delta\bullet\theta \mid (A \rightarrow \alpha\bullet\beta \in q \wedge \beta \rightarrow^* B\gamma \vee \\
&\qquad\qquad [A] \in q \wedge A \rightarrow^* B\gamma) \wedge \\
&\qquad\qquad B \rightarrow \delta\theta \wedge \\
&\qquad\qquad \exists v[v \neq \epsilon \wedge \delta\theta \rightarrow^* v] \wedge \\
&\qquad\qquad \delta \rightarrow^* \epsilon\} \\
&\quad \cup \\
&\quad \{A \rightarrow \alpha\delta\bullet\beta \mid A \rightarrow \alpha\bullet\delta\beta \in q \wedge \delta \rightarrow^* \epsilon\}
\end{aligned}
$$

It is self-evident that *repr* must be calculated from Q_0 and $K(q, X)$ instead of from their closures if efficient parser construction is required. The appropriate restatement of the algorithm calculating *repr* is straightforward.

5 Memory requirements

In this paper we have described three methods of making the (generalized) LR parsing technique applicable to hidden left-recursive grammars:

1. Apply *ϵ-elim* to the grammar before constructing the LR automaton.
2. Apply *HLR-elim* to the grammar before constructing the LR automaton.
3. Construct the ϵ-LR automaton as opposed to the LR automaton.

The last method above is derived from the first one in the sense that an ϵ-LR automaton

can be seen as a compressed LR automaton for the transformed grammar ϵ-*elim*(G). The second method is independent from the other two methods.

To investigate the static memory requirements of these methods, we have determined the number of states of the resulting automata for various grammars.

We first investigate the number of states for three kinds of characteristic grammars:

For every $k \geq 0$ we have the grammar G_1^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k c \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & b_1 \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & b_k \end{array}$$

For every $k \geq 1$ we have the grammar G_2^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k S c \\ S & \rightarrow & d \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & b_1 \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & b_k \end{array}$$

For every $k \geq 2$ we have the grammar G_3^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k c \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & S \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & S \end{array}$$

The grammars of the first group contain no left recursion. The grammars of the second group contain one occurrence of hidden left recursion, and there are k nonfalse nonterminals hiding the left recursion. The grammars of the third group contain $k-1$ occurrences of hidden left recursion, the j-th one of which is hidden by $j-1$ nonfalse nonterminals.

Figure 5 shows the numbers of states of various automata for these grammars. It also shows the numbers of states of the LR(0) automata for the original grammars. This kind of automaton does of course not terminate in the case of hidden left recursion, except if the nondeterminism is realized using cyclic graph-structured stacks, against which we raised some objections in Section 1.

These results show that the number of states is always smallest for the ϵ-LR(0) automata. A surprising case is the group of grammars G_3^k, where the number of states of ϵ-LR(0) is 6, regardless of k, whereas the numbers of states of the LR(0) automata for ϵ-*elim*(G) and *HLR-elim*(G) are exponential and quadratic in k, respectively.

In the above grammars we have found some features which cause a difference in the number of states of the automata constructed by the mentioned four methods. The results suggest that ϵ-LR parsing is more efficient in the number of states for grammars containing more hidden left recursion.

The number of states of LR and ϵ-LR automata is however rather unpredictable, and therefore the above relations between the number of states for the four methods may deviate dramatically from those in the case of practical grammars.

Practical hidden left-recursive grammars do however not occur frequently yet in natural language research. The reason is that they are often considered "ill-designed" (Nozohoor-Farshi, 1989) as they cannot be handled using most parsing techniques.

Fortunately, we have been able to find a practical grammar which contains enough hidden left recursion to perform a serious comparison. This grammar is the context-free part of the Deltra grammar, developed at the Delft University of Technology (Schoorl — Belder, 1990). After elimination of the occurrences and definitions of all predicates, this grammar contains 846 rules and 281 nonterminals, 120 of which are nonfalse. Hidden left recursion occurs in the definitions of 62 nonterminals. Rules are up to 7 members long, the average length being about 1.74 members.

The numbers of states of the automata for this grammar are given in Figure 5. These data suggest that for practical grammars containing much hidden left recursion, the relation between the numbers of states of the four

Method of construction	G_1^k $(k \geq 0)$	G_2^k $(k \geq 1)$	G_3^k $(k \geq 2)$	G_{Deltra}
LR(0) for G	$2 \cdot k + 3$	$2 \cdot k + 5$	$2 \cdot k + 2$	855
LR(0) for ϵ-*elim*(G)	$2^{k+1} + k + 1$	$3 \cdot 2^k + k + 1$	$2^{k+1} + 2$	1430
LR(0) for *HLR-elim*(G)	$2 \cdot k + 3$	$\frac{1}{2} \cdot k^2 + 4\frac{1}{2} \cdot k + 3$	$\frac{1}{2} \cdot k^2 + 2\frac{1}{2} \cdot k + 1$	1477
ϵ-LR(0) for G	$2 \cdot k + 3$	$k + 6$	6	709

Figure 5: The numbers of states resulting from four different methods of constructing LR and ϵ-LR automata.

different automata is roughly the same as for the three groups of small grammars G_1^k, G_2^k, and G_3^k: the LR(0) automata for ϵ-*elim*(G) and *HLR-elim*(G) both have a large number of states. (Surprisingly enough, the former has a *smaller* number of states than the latter, although ϵ-*elim*(G) is about 50 % larger than *HLR-elim*(G), measured in the number of symbols.) The ϵ-LR(0) automaton for G has the smallest number of states, even smaller than the number of states of the LR(0) automaton for G.

Although these results are favourable to ϵ-LR parsing as a parsing technique requiring small parsers, not for all practical grammars will ϵ-LR automata be smaller than their traditional LR counterparts. Especially for grammars which are not left-recursive, we have found small increases in the number of states. We consider these grammars not characteristic however because they were developed explicitly for top-down parsing.

Our solution requires the investigation of the parse stack. We feel however that this does not lead to deterioration of the time complexity of parsing: investigation of the stack for each reduction with some rule requires a constant amount of time. This amount of time is linear in the length of that rule, provided investigation of the symbols on the stack is implemented using a finite state automaton.

The results of our research are relevant to realization of generalized LR parsing using backtracking (possibly in combination with memo-functions) or using acyclic graph-structured stacks. Furthermore, various degrees of lookahead may be used.

We hope that our research will convince linguists and computer scientists that hidden left recursion is not an obstacle to efficient LR parsing of grammars. This may in the long term simplify the development of grammars, since hidden left recursion does not have to be avoided or eliminated.

Conclusions

We have described a solution to adapt (generalized) LR parsing to grammars with hidden left recursion. Also LR parsing of cyclic grammars has been discussed. We claim that our solution yields smaller parsers than other solutions, measured in the number of states. This has been corroborated by theoretical data on small grammars and by an empirical test on a practical grammar for a natural language.

Acknowledgements

We received kind help from Job Honig, Theo Vosse, John Carroll, and Hans de Vreught in finding a practical grammar to test our algorithms on. We acknowledge valuable correspondence with José Fortes and René Leermakers.

References

Aho, A.V. — R. Sethi — J.D. Ullman (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

Aho, A.V. — J.D. Ullman (1972). *Parsing*, The Theory of Parsing, Translation and Compiling, volume 1. Prentice-Hall.

Fortes Gálves, J. (1992). Generating LR(1) parsers of small size. In: *Compiler Construction, 4th International Conference*, LNCS 641, 16–29, Springer-Verlag.

Graham, S.L. — M.A. Harrison — W.L. Ruzzo (1980). An improved context-free recognizer. *ACM Trans. Prog. Lang. Syst.* 2(3), 415–462.

Heilbrunner, S. (1985). Truly prefix-correct chain-free LR(1) parsers. *Acta Inf.* 22, 499–536.

Knuth, D.E. (1971). Top-down syntax analysis. *Acta Inf.* 1, 79–110.

Koster, C.H.A. (1971). Affix grammars. In: Peck, J.E.L. (Ed): *ALGOL68 Implementation*, 95–109. North Holland Publishing Company.

Lang, B. (1974). Deterministic techniques for efficient non-deterministic parsers. In: *Automata, Languages and Programming, 2nd Colloquium*, LNCS 14, 255–269, Springer-Verlag.

Leermakers, R. (1992). A recursive ascent Earley parser. *Inf. Process. Lett.* 41(2), 87–91.

Leermakers, R. (1993). *The Functional Treatment of Parsing.* Kluwer Academic Publishers. To appear.

Leermakers, R. — L. Augusteijn — F.E.J. Kruseman Aretz (1992). A functional LR parser. *Theoretical Comput. Sci.* 104, 313–323.

Meijer, H. (1986). *Programmar: A Translator Generator.* PhD thesis, University of Nijmegen.

Nederhof, M.J. (1993a). Generalized left-corner parsing. In: *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, 305–314.

Nederhof, M.J. (1993b). A new top-down parsing algorithm for left-recursive DCGs. In: *Programming Languages Implementation and Logic Programming, International Workshop*, LNCS, Tallinn, Estonia. Springer-Verlag.

Nederhof, M.J. — J.J. Sarbo (1993a). Efficient decoration of parse forests. In: Trost, H. (Ed): *Feature Formalisms and Linguistic Ambiguity.* Ellis Horwood Limited.

Nederhof, M.J. — J.J. Sarbo (1993b). Increasing the applicability of LR parsing. Technical report no. 93–06, University of Nijmegen, Department of Computer Science.

Nilsson, U. (1986). AID: An alternative implementation of DCGs. *New Generation Computing* 4, 383–399.

Nozohoor-Farshi, R. (1989). Handling of ill-designed grammars in Tomita's parsing algorithm. In: *International Workshop on Parsing Technologies*, 182–192.

Pager, D. (1970). A solution to an open problem by Knuth. *Inf. and Contr.* 17, 462–473.

Schoorl, J.J. — S. Belder (1990). Computational linguistics at Delft: A status report. Report WTM/TT 90–09, Delft University of Technology, Applied Linguistics Unit.

Sippu, S. — E. Soisalon-Soininen (1990). *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing.* Springer-Verlag.

Soisalon-Soininen, E. — J. Tarhio (1988). Looping LR parsers. *Inf. Process. Lett.* 26(5), 251–253.

Thorup, M. (1992). Controlled grammatic ambiguity. Technical Report PRG-TR-2-92, Programming Research Group of Oxford University.

Tomita, M. (1986). *Efficient Parsing for Natural Language.* Kluwer Academic Publishers.

Reducing Complexity in A Systemic Parser

Michael O'Donnell

Department of Linguistics,
University of Sydney
email: mick@isi.edu

Abstract

Parsing with a large systemic grammar brings one face-to-face with the problem of unification with disjunctive descriptions. This paper outlines some techniques which we employed in a systemic parser to reduce the average-case complexity of such unification.

1 Introduction

Systemic grammar has been used in several text generation systems, such as PENMAN (Mann — Matthiessen 1985), PROTEUS (Davey, 1978), SLANG (Patten, 1986), GENESYS (Fawcett — Tucker, 1990) and HORACE (Cross, 1991). Systemics has proved useful in generation for several reasons: the orientation of Systemics towards representing language as a system of choices, the strongly semantic nature of the grammar, and the extensive body of systemic work linking discourse patterns and grammatical realisation (e.g., Halliday, 1985; Halliday — Hasan, 1976; Martin, 1992).

Parsing with systemic grammar has not, however, been as successful. To date, there have been six parsing systems using systemic grammar: Winograd (1972), McCord (1977), Cummings — Regina (1985), Kasper (1988a, 1988b, 1989), O'Donoghue (1991a, 1991b) and Bateman *et al.* (1992). However, each of these systems has been limited in some way, either resorting to a simplified formalism (Winograd — Cummings — McCord), or augmenting the systemic analysis by initial segmentation of the text using another grammar formalism (Kasper: Phrase Structure Grammar; Bateman *et al.*: Head-driven Phrase Structure Grammar; O'Donoghue: his 'Vertical Strip Grammar' (VSG)). There has not so far been a parser that parses using the full systemic formalism, without help from another formalism.

The reasons for this failure relate to those reasons which favour generation. Firstly, the orientation of systemic grammar towards choice means that the grammar is organised into a form full of disjunctions, which leads to complexity problems in parsing. Secondly, the strongly semantic content of systemic grammars (including roles such as Actor, Process and Circumstance in the grammar) leads to a structural richness which adds to the logical complexity of the task.

One result of the work in Systemic generation has been the availability of a large computational generation grammar using the systemic formalism – the Nigel grammar (Matthiessen — Mann, 1985, Matthiessen — Bateman, 1992). As this resource is available, it is desirable to use it for parsing. However, complexity problems have so far made this impossible, except by pre-parsing with another formalism.

In the last few years, we have developed a parser for Systemic grammar, particularly for use with the Nigel grammar. The parser handles the full Systemic formalism, and does not depend on another formalism for segmentation. The parser uses a bottom-up, breadth-first algorithm. A chart is used to handle some of the non-determinism.

This paper focuses on some methods we have used in the parser to reduce the complexity problems associated with using the Nigel grammar. In particular, we focus on the means used to make disjunctive unification more efficient.

Section 2 discusses the problem of disjunctive expansion, and some means of making it

more efficient at a general level. Before becoming more specific, the Systemic formalism is introduced (section 3). Section 4 explores one method of avoiding complexity – reducing the size of the disjunctive description by working with sub-descriptions rather than the whole description. Section 5 presents three ways of making expansion, when necessary, more efficient. We conclude the paper with a brief summarisation of our work.

2 Unification with Disjunctive Descriptions

Parsing with a systemic grammar involves much unification of disjunctive descriptions. The usual way to unify such is as follows:

1. Expand out the disjunctive descriptions to Disjunctive Normal Form (DNF) – a form with all disjunction at the top level of the description – a disjunction of non-disjunctive forms.
2. Unify each term of the first DNF form with each term of the other.

DNF expansion of a description is however an expensive task – the process takes exponential time in the worst case (Kasper — Rounds, 1986). Space is also a problem – DNF expansion is a transformation whereby a disjunctive description is replaced with a set of descriptions each of which contains no disjunction. For a description containing a high level of disjunction, the size of the DNF form can be excessive.

Space has not however been a problem in our processing, but time has. Systemic parsing is very slow. We thus focus on means for speeding up, or avoiding, the unification process.

2.1 Avoiding Expansion

There have been proposals for unification without DNF expansion. Karttunen, for instance, has proposed an algorithm which "uses constraints on disjuncts which must be checked whenever the disjunct is modified" (Kasper, 1987, p81). However, as noted by Kasper (1987, p61), Karttunen's unification algorithm works only for a limited type of disjunctive description, and not for general disjunction as is needed in the present work.

Kasper has proposed a method of re-representing disjunctive descriptions which in some cases avoids the need for expansion. His approach separates a disjunctive description into two parts – a *definite* component (which contains no disjunction), and an *indefinite* component (containing the disjunctive information of the description). A unification process can first check whether the definite components of two descriptions unify, and only proceeds to unify the indefinite components if the definite components unify successfully. The unification of the indefinites is avoided if the unification of the definites fails.

2.2 Delaying disjunctive expansion until necessary

The Kasper-Rounds form also allows us to delay expansion until a later time. When two descriptions are unified, only the definite components need to be checked for compatibility. The result of a Kasper-Rounds unification contains the indefinite descriptions from both descriptions without expansion. At some point in the processing it may be necessary to resolve the indefiniteness, and the disjunctive components are then expanded. However, in many cases, the definite component of the description may become inconsistent before this is necessary, expansion is thus avoided.

2.3 When expansion is necessary, expand efficiently

If DNF-expansion is required, then it should be performed as efficiently as possible. We here discuss some methods to achieve this goal:

1. **Reducing the disjunctiveness of the description**: By reducing the extent of the description, we reduce the amount of disjunction to be expanded, and thus speed up the expansion process. We use two methods to reduce the size of descriptions:
   (a) Extracting descriptions for special-purpose: we segment the grammar description into sub-descriptions for particular purposes. We found that different parsing processes drew upon

only subsets of the grammar. Rather than working with the full grammar, sub-descriptions tailored for particular purposes can be compiled-out. These sub-descriptions are less complex to expand than the full description

(b) Register Specific Pruning: parts of the grammar which are not expected to be used in a particular set of target texts are 'pruned-out' before processing begins.

2. **Expanding Disjunctions Efficiently**: a disjunctive description may contain a number of disjunctions. Ordering the expansion of these disjunctions in particular ways can result in improved expansion times:

   (a) Multiplying together disjunctions with high likelihood of inconsistency first, thus reducing the number of terms which we continue with.

   (b) Spotting inconsistent unifications with minimum of work e.g., checking for inconsistencies between single terms before checking for inconsistencies between combinations of terms.

   (c) Using some form of structure sharing in the expansion process: in the expansion process, the same terms may be multiplied together a number of times. A form of structure-sharing, such as a parse chart, can reduce the redundancy in the expansion process.

2.4 Caching and precompilation: avoiding repeating the same expansion.

The parser makes extensive use of caching – when any expansion is calculated which is likely to be used again, the result is stored away for later reuse.

Precompilation has also been a useful technique to improve parsing efficiency. Precompilation is basically a pre-caching of all the values which might be used in the parsing process. By performing most of the DNF expansion of the grammar as a precompilation step, we avoid doing that calculation during the parsing of a sentence.

3 A Systemic Grammar

3.1 Type and Role Logic

Systemic grammar, in distinction to value-attribute grammars, distinguishes *type logic* (the classes of units) and *role logic* (the constituency and dependency relations between units). The type logic is expressed in a network, called a *system network*. The role logic is expressed as a set of constraints on the types of the grammar.

3.2 System Networks

Systemic grammar (e.g., Halliday, 1985, Hudson, 1971, Matthiessen — Mann, 1985) uses an inheritance network to organise grammatical types (or 'feature' in Systemics[1]), and their structural consequences. A Systemic inheritance network is called a *system network*.

A system network is used to organise the co-occurrence potential of grammatical types, showing which types are mutually compatible, and which are incompatible. It consists of a set of *systems*, which are sets of mutually exclusive types. There is also a *covering* relation between the types of a system, meaning that if the entry condition of the system is satisfied, then one of the types in the cover must be selected.

Figure 1 shows a system network for a simple grammar of English. It includes 11 systems, representing various grammatical distinctions, for instance, between clause and word, between transitive and intransitive clauses, or between nominative and accusative pronouns.

Each type inherits the properties of types to its left in the network. Note that the system network may be logically complex, since entry conditions (the logical condition on a system) may consist of conjunctions and disjunctions of types.

[1]Note that the term 'feature' is used distinctly from its use in most unification paradigms. In Systemics, a feature is what Functional Unification Grammar would call a value, e.g., active, transitive and noun are features.

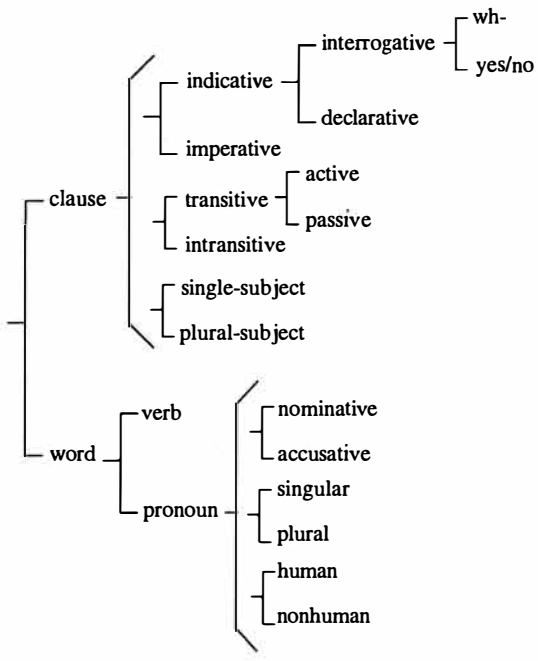


Figure 1: A partial Systemic network

3.3 Structural Templates

Types of the system network are associated with structural realisations – the structural consequence of the type. Figure 2 shows the realisations of the types in Figure 1.

```
clause:          Subject: nominative
                 Actor: human
                 Finite: finiteverb
                 Pred: lexical-verb
declarative:     Subject^Finite
yes-no:          Finite^Subject
transitive:      Object: accusative
                 Actee = [ ]
                 Pred...Object
active:          Subject/Actor
                 Object/Actee
                 Finite/Pred
passive:         Subject/Actee
                 Object/Actor
                 Pass: be-aux
                 AgentM = "by"
                 Finite/Pass
                 Pred: en-verb
                 Pass^Pred
                 AgentM^Object
intransitive:    Subject/Actor
                 Finite/Pred
single-subj:     Subject: singular
plural-subj:     Subject: plural
```

Figure 2: Realisation Rules

This grammar deals mainly with some systems involving the Subject and Object, what types of units fill these roles, and how these roles conflate with two other roles: Actor and Actee. The grammar assumes that both roles are filled by pronouns, which are either [nominative] or [accusative], [singular] or [plural], and [human] (e.g., "I", "you", "he") or [nonhuman] (e.g., "it", "that"). Only [human] pronouns can fill the Actor role of a clause.

The realisation operators used in the formalism are as follows:

Insert e.g., *Finite = [ ]*: indicates that the function *Finite* must be present in the structure.

Conflate e.g., *Modal/Finite*: indicates that the two functions *Modal* and *Finite* are filled by the same grammatical unit.

Order e.g., *Subject ^ Finite*: indicates the sequencing of functions in the surface structure. In this example, the *Subject* is sequenced directly before the *Finite*. Any number of elements can be sequenced in a single rule.

Partition e.g., *Thing... Event... End*: Another sequence operator, specifies that the appear in this order, but not necessarily immediately adjacent (linear precedence).

Preselect e.g., *Subject: nominal-group*: indicates that the *Subject* element must be filled by a unit of type *nominal-group*.

Lexify e.g., *Deict = "the"*: used to assign lexical items directly to elements of structure. Note that lexify overrides any preselect which may apply to the same element of structure.

3.4 Logical Expression of the Grammar

For the purposes of the expansion of this grammar, we re-express it in a logical formalism. Figure 3 shows Logical Form I of this grammar, including the structural constraints embedded in the form. Note that :xor indicates exclusive disjunction.

```
(:xor
 (:and clause
       Subject: nominative
```

```
      Actor: human
      Finite: finite-verb
      Pred: lexical-verb
      (:xor
        (:and declarative
              Subject^Finite)
        (:and yes-no
              Finite^Subject))
      (:xor
        (:and transitive
              Object: accusative
              Actee = [ ]
              Pred...Object
              (:xor
                (:and active
                      Subject/Actor
                      Object/Actee
                      Finite/Pred
                      (:and passive
                            Subject/Actee
                            Object/Actor
                            Pass: be-aux
                            AgentM= "by"
                            Pred: en-verb
                            Finite/Pass
                            Pass^Pred
                            AgentM^Object))
        (:and intransitive
              Subject/Actor
              Fin/Pred))
      (:xor (:and single-subject
                  Subject: singular))
            (:and plural-subject
                  Subject: plural))))
(:and word
      (:xor (:and pronoun
                  (:xor nominative
                        accusative)
                  (:xor singular
                        plural)
                  (:xor human
                        nonhuman))
            (:and verb ... ))))
```

Figure 3: Logical Form I of the Grammar

4 Extracting Sub-Grammars for particular Parsing Tasks

Rather than expanding out the whole grammar, it is more efficient to extract out subsets of the grammar, to be used for particular tasks in parsing. In our systemic parser, the description is used for three purposes:

1. Path Unification: checking that two type-paths can unify,

2. Predicting What Comes Next: seeing which function-bundle(s) can come next in the structure e.g., we have just analysed Subject/Actor ^ Fin/Mod, and want to predict what function-bundle can occur next in the structure.

3. Function-Bundle Assignment: seeing what function-bundle a given constituent can fill, e.g., we have just parsed a nominal group, and want to see what function-bundles it can be the filler of.

Each of these uses makes only partial use of the grammar description. Thus, rather than expanding out the entire grammar, we can simplify the process by extracting out sub-grammars, one for each of these applications. Since the size of each sub-grammar is smaller, the complexity problem is reduced. This section looks at these three sub-descriptions in more detail.

4.1 Separating Type Logic from Role Information

It has proved useful to separate the type logic component of the grammar from the role logic. The two logic components have different patterns of use – type logic is used to test whether two partial type-paths can unify. We never try to unify a partial type description with the type grammar as a whole. The type-logic component of the grammar thus does not need to be DNF-expanded.

The role logic, on the other hand, does need to be expanded. We expand the role-logic component to produce a set of non-disjunctive structure rules which can be applied during parsing (sometimes termed 'chunking').

```
(:and
 ;1. Type Logic Component
 (:xor (:and clause
             (:xor declarative yes-no)
             (:xor (:and transitive  (:xor active passive))
                   intransitive)
             (:xor single-subject plural-subject))
       (:and word
             (:xor (:and pronoun (:xor nominative accusative)
                                 (:xor singular plural)
                                 (:xor human nonhuman))
                   (:and verb ... ))))

 ;2. Role Logic Component
  (:and  (:implies clause (:and Subject: nominative
                                Actor: human
                                Finite: finite-verb
                                Pred: lexical-verb))
         (:implies declarative Subject^Finite)
         (:implies yes-no  Finite^Subject)
         (:implies transitive  (:and Object: accusative
                                     Actee: [ ]
                                     Pred...Object))
         (:implies active (:and Subject/Actor
                                Object/Actee
                                Finite/Pred))
         (:implies passive (:and Subject/Actee
                                 Object/Actor
                                 Pass: be-aux
                                 AgentM= "by"
                                 Pred: en-verb
                                 Finite/Pass
                                 Pass^Pred)
                                 AgentM^Object))
         (:implies intransitive  (:and Subject/Actor
                                       Fin/Pred))
         (:implies single-subject Subject: singular)
         (:implies plural-subject Subject: plural)))
```

Figure 4: Logical Form II of the Grammar

Thse two components of the description have different properties: type logic is acyclic, while role logic is potentially cyclic. Type logic is constrained such that types are always in disjoint coverings (which allows efficient negation), while role logic doesn't have this constraint.

Because of these differences in properties and uses, it has proved efficient to treat these two logics separately. Logical Form I of the systemic grammar provided in Figure 3 can be re-represented in the equivalent Logical form II shown in Figure 4, separating out the type and role logic.

4.1.1 Unification of Type Descriptions

The parser uses the type-logic component of this grammar without fully expanding it. Partial expansion, however, is performed, whereby the *type-path* (the logical-entailment of a system, i.e., the logical expression of types leading back to the

root of the network)[2] is pre-compiled for each system.[3] The negation of each type in the system is also pre-compiled, which speeds up unification involving negation of types.

Type-paths are represented in the form proposed by Kasper (1987), and his unification algorithm is used when two type-paths are unified. The main use of the type-logic component is checking the compatibility of two types or type-paths.

Type logic has thus been simplified using three strategies:

1. Separating from Role Logic
2. Using Kasper's 'delayed expansion' technique.
3. Precompiling each system's logical entailment, and the negation of types.

Because of these methods, unification of type-paths using even quite complex grammars operates quite quickly.

4.2 Function Assignment

Another use made of the grammatical description in parsing is to assign a set of structural roles to a unit. The set of roles a unit fills is called in Systemics the *function-bundle* of the unit. The systemic formalism allows each unit to be assigned multiple functions. For instance, using the NIGEL grammar, 'the cat' in "the cat scratched the woman" would be assigned the function-bundle Subject/Agent/Actor/Theme. The possibility of a unit serving multiple functions is a major source of complexity in systemic parsing.

Assigning function-bundles to a unit is one of the tasks in systemic parsing. For instance, assume we have just parsed a pronoun "he", assigning it a type-path:

```
(:and word:pronoun:nominative:human:singular)
```

Now, we wish to find what function-bundles the pronoun can serve at a higher level. One result could be:

```
                          [clause:transitive]
[pronoun]                   ______|___...
    |          =>                 |
  "he"                      Subject/Actor
                              [pronoun]
```

This process draws upon three parts of our grammar:

- Preselection and Lexify rules: used to discover what functions different units can fill.
- Conflation rules: used to discover which functions a unit can serve simultaneously, and thus, which of the preselection and lexify rules can combine.
- The Type Logic: to show which of these preselection, lexify and conflation rules are systemically compatible.

Since we have already set up the type-logic for path unification, we can draw upon that resource as needed. We do not need to include the type-logic in the sub-description for the function-assignment process.

4.2.1 Extracting the relevant description

For the function-assignment process, we do not need all of the role logic description. We can select out only those rules involving preselection, lexify, and conflation. See Logical Form III in Figure 5.

```
(:and (:implies clause
         (:and Subject: nominative
               Actor: human
               Finite: finite-verb
               Pred: lexical-verb))
      (:implies transitive
         Object: accusative)
      (:implies active
         (:and Subject/Actor
               Object/Actee
               Finite/Pred))
      (:implies passive
         (:and Subject/Actee
               Object/Actor
               Pass: be-aux
```

[2]Note that since entry conditions of systems can be logically complex, the path itself can contain disjunctions and conjunctions.

[3]Paths are stored with systems rather than types, since the path of all types in a system are identical.

```
            AgentM= "by"
            Pred: en-verb
            Finite/Pass))
    (:implies intransitive
       (:and Subject/Actor
             Fin/Pred))
    (:implies single-subject
          Subject: singular)
    (:implies plural-subject
          Subject: plural)))
```

Figure 5: Logical Form III: The Function Assignment Sub-Description

4.2.2 Implications Out

We next put this description into a form more suitable for DNF-expansion. Note that implication can be re-expressed using disjunction, conjunction and negation:

```
(:implies a b)  is-equivalent-to
       (:xor (:and a b) (:not a))
```

Using this rule, we can re-express the logical form III as Logical Form IV, as shown in Figure 6.

```
(:and (:xor (:and clause
                  Subject: nominative
                  Actor: human
                  Finite: finite-verb
                  Pred: lexical-verb)
            (:not clause))
      (:xor (:and transitive
                  Object: accusative)
            (:not transitive))
      (:xor (:and active
                  Subject/Actor
                  Object/Actee
                  Finite/Pred))
            (:not active))
      (:xor (:and passive
                  Subject/Actee
                  Object/Actor
                  Pass: be-aux
                  AgentM: "by"
                  Pred: en-verb
                  Finite/Pass)
            (:not passive))
      (:xor (:and intransitive
                  Subject/Actor
                  Fin/Pred)
            (:not intransitive))
      (:xor (:and single-subject
                  Subject: singular)
            (:not single-subject))
      (:xor (:and plural-subject
                  Subject: plural)))
            (:not plural-subject)))
```

Figure 6: Logical Form Form IV: The Function Assignment

4.2.3 Expansion to DNF

Simple algorithms exist to expand Logical Form IV into DNF (see section 5.1). A small part of the result appears in Logical Form V of the grammar, shown in Figure 7.

```
(:xor
  (:and clause transitive active
        single-subject
        Subject/Actor: (:and nominative
                             human singular)
        Object/Actee: accusative
        Finite/Pred: (:and verb finite-verb
                           lexical-verb))
  (:and clause transitive
        active plural-subject
        Subject/Actor: (:and nominative
                             human plural)
        Object'Actee: accusative
        Finite/Pred: (:and verb finite-verb
                           lexical-verb))
  etc...
```

Figure 7: Logical Form Form V: The Function Assignment Sub-Description in DNF

The order of worst-case complexity of the expansion to DNF is easily calculated – it is simply two to the power of the number of disjunctions, which is equal to the number of types which have realisation rules of type conflation, insertion, or preselection.

By opting to expand only subsets of the whole grammar, we have reduced the complexity of the description, since the size of n for this sub-description is smaller than for the whole description. However, for a real-sized grammar such as NIGEL, the size of n is still large.

4.2.4 Re-expression in terms of Function Bundles

From the DNF-form of this description, we can extract out partial-descriptions for each function bundle. We now re-express this logical form in

terms of the type constraints on each function-bundle, including both the constraint on the type of unit the function-bundle can be part of (the 'parent-constraint'), and the constraint on the filler of the function-bundle (the 'filler-constraint'). We show this as a set of triplets, of the form:

```
( <parent-types>
  <function-bundle>
  <child-types> )
```

```
1. ( (:and clause transitive
           active single-subject)
     Subject/Actor
     (:and nominative human singular)))

2. ( (:and clause transitive
           active single-subject)
     Object/Actee
     accusative)))

3. ( (:and clause transitive
           active plural-subject)
     Subject/Actor
     (:and nominative human plural)

4. ( (:and clause transitive
           active plural-subject)
     Object/Actee
     accusative)

5.  ( (:and clause transitive
            active singular-subject
      Finite/Pred
      (:and verb finite-verb lexical-verb))
etc....
```

This representation can now be used to assign function-bundles A unit can take on a function-bundle if it can unify with the filler-constraint on the function-bundle.

For the instance we started with, "he", with types: `(:and pronoun nominative human singular)`, only one triplet would unify. We could thus posit structure for our unit:

```
[clause:transitive:active:single-subject]
    ..----------|----------...
                |
         Subject/Actor
                |
 [ pronoun:nominative:human:singular ]
                |
              "he"
```

Note that we have also gained information about the types of the parent-unit of which the unit is a constituent.

4.2.5 Reducing the number of Rules

Note that there is another simplification we can make to the triplet list. We can take all triplets with identical function bundle and child-type specification, and join them. The parent-types slot is replaced with the disjunction of the two parent-type slots. Thus, elements 2 and 4 above become a single item. This process reduces the number of rules to apply:

```
2,4. ( (:and clause transitive active)
       Object/Actee
       accusative)))
```

4.3 Predicting What Comes Next

Another process we use in parsing involves the prediction of what function-bundles can come next in a partially completed structure. With a systemic grammar, this process requires:

- Ordering and Partition rules: to see which function can come next.
- Conflation rules: to see which functions can conflate with the function predicted to come next.
- The type logic: to show which of these ordering, partition and conflation rules are systemically compatible.

The processing of this sub-description, and any others, is exactly the same as for function-assignment.

1. Extract from the role logic description the relevant realisation rules;
2. Replace implications with disjunction and negation;
3. Expand out the grammar;
4. Index the rules in a form useful for the processing.

4.4 Register Restriction

Another means of reducing the overall complexity of the descriptions involves eliminating from the grammar parts which are unlikely to be utilised in the target texts. In systemic terms, we apply register restrictions to the grammar.

For example, in a domain of computer manuals, the description of interrogative structures is not likely to be drawn upon.[4] By eliminating this sub-description, we reduce the degree of disjunction in the whole description, and thus speed up the parsing of the forms which do appear in the text.

The method of deriving the register-restrictions was as follows:

1. We parsed by hand[5] a chapter of the computer manuals we were attempting to parse, building up a register-profile of our target texts.

2. An automatic procedure then extracted out all the grammatical types which occur in these sentences.

3. The process used this information to discover the types *not* occurring in the sample.

4. The process then eliminated these types and their realisations from the description.

We were thus left with a restricted grammar which was capable of parsing the sentences in the sample, and also parsing many which were not in the sample (under the assumption that the grammatical forms in the sample were representative of the forms found in the manual as a whole). We reduced the size of the grammar by approximately 60% using this method.

4.5 Summary

By extracting out sub-descriptions from the full description, we reduce the complexity of the description-to-be-expanded.

5 Improving the Efficiency of Expansion

Section 4 has proposed techniques which reduce the size of the description which needs to be expanded. However, for large-sized descriptions, the expansion is still complex. This section briefly explores two methods which increase the efficiency of the expansion process. If we can't avoid full expansion, then at least we can make the expansion process more efficient.

5.1 "Structure Sharing" in Expansion

This section assumes a disjunctive description of the following form:

```
(:and (:xor A B) (:xor C D) (:xor E F) )
```

Logical form V introduced above was of this form. Much of the pre-processing in the parser involves the DNF-expansion of disjunctions in this form.

5.1.1 Full Expansion

The brute force method for expanding this form involves:

1. Find all combinations of terms, taking one term from each disjunction.

2. Test compatibility of each combination, eliminating combinations which are internally inconsistent.

Step 1 of this process produces the following DNF form:

```
(:xor (:and A C E) (:and A C F)
      (:and A D E) (:and A D F)
      (:and B C E) (:and B C F)
      (:and B D E) (:and B D F) )
```

[4]Note that some of the forms we restrict through register restriction may actually appear in any one text, although quite rarely. We are trading off between speed for the majority of sentences, and ability to parse all sentences in a text.

[5]The hand-parsing is really computer-assisted, – a tool was developed to traverse the system network for each sentence (and each constituent of the sentence) asking the human which feature was appropriate for the target string. This process guaranteed that the human-analysis conformed to the computer grammar.

The problem with this approach is with the incompatibility checking – the same checks will be repeated over and over again. For instance, the incompatibility check between A and C is repeated twice: (:and A C E) and (:and A C F). This repetition occurs for every pair of terms in the conjuncts. The problem gets worse exponentially as we add more disjuncts.

To avoid this redundancy, we need something like a chart in parsing, a method to record the results of each unification and thus avoid repeating any unification.

Unfortunately, DNF expansion is not quite like parsing. We can test the consistency between any two pairs of terms (for instance A and C in the above), but we also need to know about the consistency of terms in combination e.g., the pairs: A&C, A&E and C&E may be consistent, but the combination A&C&E may not be.

The rest of this section describes two techniques which allow some redundancy reduction, sometimes known as structure-sharing.

5.1.2 Tree Organisation of Expansion

The disjunctive description above can easily be re-represented in the form below:

```
(:and (:and (:xor A B)
            (:xor C D))
      (:xor E F))
```

The process here involves expanding out the first two disjunctions, eliminating inconsistent results, and then expanding the result out with the next disjunction. The incremental expansion is illustrated in Figure 8.

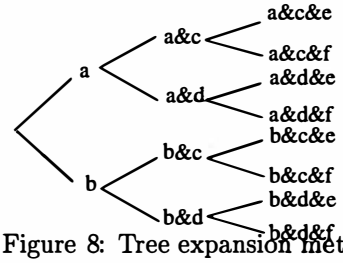


Figure 8: Tree expansion method

This method is more efficient than the full expansion method, since:

- Some terms, such as a&c, a&d etc. are unified only once. However note that terms e and f are still involved in multiple products.
- the failure of a combination of terms early in the unification process eliminates a large number of expansions by the end of the process.

5.1.3 Binary Organisation of Expansion

A third approach aims at maximising the degree of 'sharing' unifications in the expansion. The disjunctions in the description are split into pairs, and unified. The results of these unifications are then unified in the same pair-wise manner. This expansion for a conjunction of four disjunctions is shown in Figure 9.

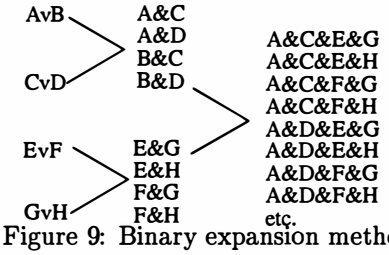


Figure 9: Binary expansion method

The advantage of this approach is that we are maximising the amount of structure-sharing in the unification.

5.1.4 Comparison of Expansion Approaches

We compared the number of unifications which take place using each of these methods for various numbers of disjunctions (all disjunctions having two disjuncts).

One can see from Table 1 that the worst-case score for the full expansion method is far worse than the other methods. It is not a practical method.

Comparing the worst-case for the 'tree' and the 'binary' expansion method, we see that the binary method clearly comes out better, by around 50%.

We also did a simulation to check an average case score, since the worst-case score doesn't take into account that many later unifications are avoided when early unification proves inconsistent. We found that while the binary method still seems superior, in some instances the tree method requires fewer unifications. More work is needed here.

N	Full	Tree	Binary
11	20480	4092	2364
12	45056	8188	4424
13	98304	16380	8448
14	212992	32764	16780
15	458752	65532	33236
16	983040	131068	66144
17	2097152	262140	133528
18	4456448	524284	266660
19	9437184	1048572	526956
20	19922944	2097148	1053304

Table 1: Worst-case comparison

5.2 Ordering Incompatible Disjunctions First

When using either the Tree or Binary expansion methods, fewer unifications will be required if we place the disjunctions with the greatest chance of inconsistency first. In a sense, we are pruning inconsistent branches of the expansion tree 'at the root'.

In the systemic parser, several heuristics have been used to group disjunctions which are most likely to produce the fewest cross-products, and perform these first.

One possible method for utilising this phenomenon is:

1. Separate the disjunctions into sub-sets which maximise likelihood of incompatibility between rules inside the sub-expressions.
2. Expand out the disjunctions inside each sub-set. The results of each sub-set are cached so they need only be expanded once.
3. Expand out the results of (2) against each other.

5.3 Avoiding Expansion of Incompatible Terms

Sometimes, it is possible to tell without full unification that a set of rules will not unify with another set. For instance, assume a larger grammar than the one we have been using, a grammar which includes clauses, nominal-groups[6], prepositional phrases, adverbial phrases and words. These categories are all types in the system network, just like any other types.

Since these types won't unify with each other, we can also know that types which inherit from one of these basic types will not unify with the the sub-types of another basic type. We thus do not need to try to unify descriptions which differ in their basic type. If we split any disjunctive description into sub-components for each basic type, we know a priori that there is no unification between these sub-components.

Before trying any of the expansion techniques outlined in this paper, the whole grammar is segmented into sub-descriptions, one for each of these basic types. The complexity of the expansion of each of these sub-grammars is less than for the grammar as a whole.

Other principles can be used to locate sets of rules which will not unify. These can be applied also.

6 Conclusion

While the techniques outlined here have been applied in ways particular to a systemic grammar, and for a particular implementation, there are principles behind the re-representations which are general to all implementations:

1. Avoid DNF-expansion where possible, as in Kasper's unification algorithm.
2. Delay expansion to a later time – information gained later may show the description to be inconsistent in the definite component.
3. When expansion is necessary,
   (a) Try to extract out sub-descriptions which can be used, rather than expanding the entire grammar.
   (b) Expand out first disjunctions which are most likely to conflict, since this will reduce the total number of terms which will need to be multiplied.
   (c) Avoid expanding terms that can be known to be incompatible.

[6]Systemics prefers the term 'nominal-group' over the equivalent term 'noun-phrase'.

As a result of the application of these techniques (and others not here mentioned), we have been able to implement a parsing system which parses using a large systemic grammar.

1. We start with the Nigel grammar, as used in the Penman Generation System, slightly modified for parsing purposes.
2. This grammar is then reduced by applying register-restrictions, leaving a less complex grammar, but a grammar which still handles the bulk of the phenomena in the target texts.
3. Sub-descriptions of the grammar tailored for particular processes are then extracted, and expanded out as a precompile step, producing a set of 'chunks' which can be used in parsing. This expansion takes approximately 2 minutes using Sun Common Lisp on a Sun Sparc II.
4. The 'chunked' grammar is then used to parse sentences. On the above-mentioned platform, parsing a sentence like "A user-password is a character string consisting of a maximum of eight alpha-numeric characters." took 35 seconds to parse[7]. This parser is slow, compared to most non-systemic parsers, but is far faster than the parser would be without the methods outlined here.

Future work will attempt to reduce this parsing time. Three directions are being followed:

- Streamlining the parsing process to further reduce the parsing time.
- Moving more processing to the pre-compilation stage.
- Reducing the complexity of the description without reducing its coverage.
- Incorporating heuristics to resolve ambiguities without full expansion.

Acknowledgements

The parser discussed in this paper was partially developed in the Electronic Discourse Analyser project, funded by Fujitsu (Japan). The development was aided by discussions with the members of that project: Christian Matthiessen, John Bateman, Zeng Licheng, Guenter Plum, Arlene Harvey and Chris Nesbitt.

Thanks to Cécile Paris for profuse commenting on this paper, and teaching me Latex, and to Vibhu Mittal, who solved the trickier Latex bugs.

[7]Note that when the parser is given a less complex systemic grammar, the parsing time is under two seconds for this sentence.

References

Bateman, John — Martin Emele — Stefan Momma (1992) "The nondirectional representation of Systemic Functional Grammars and Semantics as Typed Feature Structures" in *Proceedings of COLING-92*, Volume III, Nantes, France, 916-920.

Benson, J. — W. Greaves (eds.) (1985) *Systemic Perspectives on Discourse*, Volume 1. Norwood: Ablex.

Cross, Marilyn (1991) *Choice in Text: A Systemic-Functional Approach to Computer Modelling of Variant Text Production*, Ph.D. thesis submitted June 1991, Macquarie University.

Cummings, Michael — Al Regina (1985) "A PROLOG parser-generator for Systemic analysis of Old English Nominal Groups", in Benson and Greaves, 1985.

Davey, Anthony (1978) *Discourse Production: a computer model of some aspects of a speaker*, Edinburgh: Edinburgh University Press. Published version of Ph.D. dissertation, University of Edinburgh, 1974.

Fawcett, Robin P. — Gordon H. Tucker (1990) "Demonstration of GENESYS: a very large semantically based Systemic Functional Grammar". In *Proceedings of the 13th Int. Conf. on Computational Linguistics (COLING '90)*.

Halliday, M. A. K. (1985) *Introduction to Functional Grammar*, London: Edward Arnold.

Halliday, M. A. K. — R. Hasan (1985) *Cohesion in English*, London: Longman.

Hudson, R.A. (1971) *English Complex Sentences*, North-Holland.

Kasper, Robert (1986) "Systemic Grammar and Functional Unification Grammar" In Benson, J. and Greaves, W., *Selected Papers from the 12th International Systemics Workshop*, Norwood, N.J: Ablex.

Kasper, Robert (1987a) *Feature Structures: A logical Theory with Application to Language Analysis*, PH.D. dissertation, University of Michigan

Kasper, Robert (1987b) "A Unification Method for Disjunctive Feature Descriptions" in *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, held July 6-9, 1987 Stanford, California.

Kasper, Robert (1988a) "An Experimental Parser for Systemic Grammars", *Proceedings of the 12th Int. Conf. on Computational Linguistics*, Budapest: Association for Computational Linguistics.

Kasper, Robert (1988b) "Parsing with Systemic Grammar", Mimeo.

Kasper, Robert (1989) "Unification and Classification: An Experiment in Information-Based Parsing" In *Proceedings of the International Workshop on Parsing Technologies*, pages 1-7, CMU, Pittsburgh.

Kasper, Robert (1990) "Performing Integrated Syntactic and Semantic Parsing Using Classification" paper presented at Darpa Workshop on Speech and NL Processing, Pittsburgh, June 1990.

Kay, Martin (1979) "Functional Grammar" in *Proceedings of the Fourth Annual Meeting of the Berkeley Linguistics Society*.

Kay, Martin (1985) "Parsing In Functional Unification Grammar" in Dowty D., L. Karttunen , and A. Zwicky, (Eds): *Natural Language Parsing*, Cambridge University Press, Cambridge, England.

Mann, W. C. and C. I. M. Matthiessen (1985) "Demonstration of the Nigel Text Generation Computer Program". in Benson and Greaves, 1985

Martin, James (1992) *English Text: System and Structure*, Amsterdam: Benjamins.

Matthiessen, C. I. M. and W. C. Mann (1985) "NIGEL: a Systemic Grammar for Text Generation" in Benson and Greaves, 1985

Matthiessen, C. I. M. and J. Bateman (1992) *Text Generation and Systemic Functional Linguistics: Experiences from English and Japanese*. London: Pinter Publishers.

McCord, Michael (1977) Procedural Systemic Grammars in *Int. J. Man-Machine Studies*, 9, 255-286, London: Academic Press.

Mellish, Chris (1988) "Implementing Systemic Classification by Unification", *Computational Linguistics*, Vol. 14, Number 1, Winter 1988.

O'Donoghue, Tim F. (1991a) "The Vertical Strip Parser: A lazy approach to parsing" Research Report 91.15, School of Computer Studies, University of Leeds, Leeds, UK.

O'Donoghue, Tim F. (1991b) "A Semantic Interpreter for Systemic Grammars" in *Proceedings of the ACL Workshop on Reversible Grammars*, University of California at Berkeley, June 1991.

Patten, Terry and Graeme Ritchie (1986) "A formal model of Systemic Grammar", paper presented at 3rd International Workshop on Language Generation, Nijmegen, August 19-23, 1986.

Patten, Terry (1986) *Interpreting Systemic Grammar as A Computational Representation: A Problem Solving Approach to Text Generation*, Ph. D. dissertation, University of Edinburgh.

Winograd, Terry (1972) *Understanding Natural Language*. New York: Academic Press.

Generalized LR parsing and attribute evaluation

Paul Oude Luttighuis and Klaas Sikkel

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, the Netherlands
email: {oudelutt|sikkel}@cs.utwente.nl

Abstract

This paper presents a thorough discussion of generalized *LR* parsing with simultaneous attribute evaluation. Nondeterministic parsers and combined parser/evaluators are presented for the *LL*(0), *LR*(0), and *SKLR*(0) strategies. *SKLR*(0) parsing occurs as an intermediate strategy between the first two. Particularly in the context of simultaneous attribute evaluation, generalized *SKLR*(0) parsing is a sensible alternative for generalized *LR*(0) parsing.

1 Introduction

Natural language theory and programming language theory have a common foundation in formal language theory. In particular, context-free grammars find broad application in both fields. Yet, whereas programming language theory allows for restriction of the class of applicable grammars in order to obtain more efficient parsers, natural language theory typicaly demands parsing algorithms for general context-free grammars.

Semantic issues are generally handled by different formalisms in both theories. While attribute grammars (AGs) are widely used in programming language circles, natural language semantics are dealt with by e.g. feature-structure grammars. A well-known problem in programming language theory is the evaluation of the attributes of an AG *during* parsing. Attribute evaluation during parsing allows for refraining from storing the parse tree and can therefore be space efficient. Moreover, attribute values, which have been computed during parsing, may be used to control the parser, for instance by (partly) resolving parsing conflicts.

This paper reports on such attribute evaluation during parsing. The parsing algorithm used, however, is a typical natural language parsing algorithm: generalized LR parsing, also called Tomita's algorithm. By combining Tomita's algorithm with simultaneous attribute evaluation, programming language theory may benefit from the virtues of Tomita's algorithm, while natural language theory may benefit from the efficient techniques for simultaneous attribute evaluation, found in programming language theory.

Tomita's algorithm provides a clever deterministic implementation of the nondeterminism occurring in parsing non-*LR* grammars in an *LR* fashion. Attribute evaluation during deterministic *LR* parsing is a non-trivial problem and has raised considerable interest in literature on programming language implementation. On the other hand, attribute evaluation during deterministic *LL* parsing is simple, provided we restrict ourselves to the use of L-attributed AGs only, which is what we will do in the entire paper. L-attributed AGs are such that dependencies between attribute values do not flow but from left to right in the parse tree.

Because of this observation, we structured our paper as follows. We start with discussing nondeterministic *LL* parsing. By means of an intermediate step (yielding a technique called *SKLR* parsing), this is transformed into *LR* parsing. Then, the nondeterministic *LL* parser is enhanced with simultaneous attribute evaluation and the same transformation steps are used to obtain attribute evaluation during nondeterministic *LR* parsing. The most severe difficulties occur in the first transformation step.

The following principles characterize our dis-

cussion.

- Generality. No determinism is required beforehand.
- Finiteness. Except for the attribute domains, the data and control structures of our algorithms must be finite.
- Static evaluation. There must be no need at all to check at evaluation time whether an attribute instance has been evaluated yet. An attribute instance must be evaluated at the moment the parser enters the state with which the instance is associated.

Within the bounds of these principles, we push the combination of attribute evaluation and parsing to its very limits. Efficiency is not our first concern. We refrain from using dynamic evaluation techniques, by which the evaluation of attribute values can be postponed. Efficiency considerations may afterwards be used to obtain efficient implementations.

2 Context-free grammars and attribute grammars

We assume that the reader is familiar with context-free grammars (CFGs). The CFGs in this paper always have a production of the form $S \rightarrow \#X\$$ such that S does not occur in any other production. *Left-recursive* CFGs have a nonterminal A and a string α of grammar symbols for which $A \Rightarrow_G^+ A\alpha$. *Hidden-left-recursive* CFGs have a nonterminal A and a strings α, β of grammar symbols for which $A \Rightarrow_G^+ \alpha A \beta$ and $\alpha \Rightarrow_G^+ \varepsilon$.

We will not present a formal definition of attribute grammars (AGs). An AG is based on a CFG. Every grammar symbol carries a series of *attributes* of a certain type. In a parse tree, therefore, *instances* of these attributes occur. The attribute instances in one production in the parse tree are functionally dependent on one another. In the AG, this dependence is specified by a set of *semantic rules*, associated with that production. The function of an *attribute evaluator* is to assign values to the attribute instances according to these semantic functions.

We distinguish two kinds of attributes: *synthesized* and *inherited* ones. Synthesized attributes of a symbol depend on other attributes in the production *below* that symbol, whereas inherited attributes depend on attributes in the production *above* it.

In order to simplify the discussion, we use a special form of AGs, caled *untyped L-attributed AGs*, or *ULAGs*. In fact, an ULAG is an AG,

- which is L-attributed. This means that inherited attribute occurrences of right-hand side symbols cannot depend on synthesized attribute occurrences of right-hand symbols to their right.
- in which all attributes have the same type;
- in which every symbol has exactly one inherited and exactly one synthesized attribute.
- in which any attribute occurrence depends on *all* used attribute occurrences to its left in that production.

The first restriction is the most severe of all. The other three are easily dealt with by simple rewrites of the AG.

3 $LL(0)$ and $LR(0)$ parsers

This section present a nondeterministic $LL(0)$ parser and transform it, via an $SKLR(0)$ parser into an $LR(0)$ parser.

3.1 L-parsers

This subsection gives a short description of the parser model, called *L-parser*, used in this paper, without presenting a formal definition.

The core of an L-parser is a transition relation between *instantaneous descriptions*. Such an instantaneous description consists of a stack contents, which is a string of states, and a (remaining) input, which is a string of grammar symbols. Operation starts with a special stack contents, consisting of a special state, the *initial state* and the input string. This instantaneous description may be turned into other ones by means of the transition relation. Operation stops if the top of the stack is another special state, the *final state*.

As opposed to more common parser models, an L-parser is allowed to pre-append symbols to the remaining input during reduction steps. Our reduction steps are such that they pre-append the lhs of the associated production to the input, instead of shifting it immediately. This shift is performed by a compulsory subsequent shift step. Advantages are that

- It simplifies the description of parsers.
- It facilitates the definition of attribute evaluation during parsing.
- It enables parsing of arbitrary sentential forms, instead of just terminal strings.
- It nicely generalizes to parsers for context-sensitive grammars.

3.2 Dotted rules

LR parsing was introduced by Knuth (1965). Although *LL* and *LR* parsing can be elegantly described as being each other's duals (Sippu and Soisalon-Soininen, 1990), they can both be seen as (implicitly) performing a depth-first left-to-right walk over the (virtual) parse tree.

Positions in this walk are indicated by *dotted rules*. In essence, a dotted rule is a production together with some position in its rhs. This position is rather *between* symbols that *at* them. A dotted rule $[A \to X_1 \ldots X_i \bullet X_{i+1} \ldots X_n]$, where $n \geq 0$ and $0 \leq i \leq n$, indicates that the parser's current position during the tree walk is in an application of $A \to X_1 \ldots X_n$ at the position between X_i and X_{i+1}. Dotted rules of the form $[A \to \bullet\alpha]$ are called *starting* dotted rules, others are called *proper* dotted rules.

During a tree walk, three kinds of steps occur: production steps, shift steps, and reduction steps. Production steps are steps down in the parse tree. They always leave from a dotted rule of the form $[A \to \alpha_1 \bullet B\alpha_2]$ and arrive at a dotted rule of the form $[B \to \bullet\beta]$. Shift steps are steps to the right, leaving from a dotted rule of the form $[A \to \alpha_1 \bullet X\alpha_2]$ and arriving at one of the form $[A \to \alpha_1 X \bullet \alpha_2]$. Finally, reduction-shift steps are steps up in the parse tree and they leave from a dotted rule of the form $[B \to \beta\bullet]$ to arrive at one of the form $[A \to \alpha_1 B \bullet \alpha_1]$. See Figure 1.

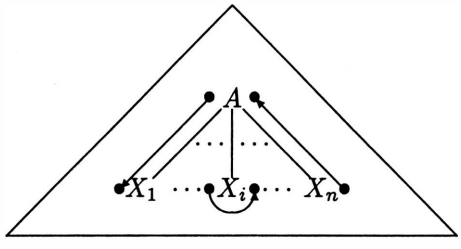


Figure 1: Production, shift, and reduction-shift steps in a parse tree.

3.3 $LL(0)$ parsers

Dotted rules are the states of our $LL(0)$ parsers. The initial state is $[S \to \# \bullet X\$]$, the final one $[S \to \#X\$\bullet]$. At any moment, the $LL(0)$ parser may perform

- a production step. In this case, the current state must be of the form $[A \to \alpha_1 \bullet B\alpha_2]$. Any dotted rule of the form $[B \to \bullet\beta]$, can be the next current state, and hence be pushed onto the stack. The remaining input is not changed;
- a shift step, in which case the current state must be of the form $[A \to \alpha_1 \bullet X\alpha_2]$ and the remaining input must start with an X. The new current state is $[A \to \alpha_1 X \bullet \alpha_2]$ and is pushed onto the stack. The X leading the remaining input is removed;

- a reduction-shift step. In this case, the current state must be of the form $[A \to X_1 \ldots X_n \bullet]$. Now, by the nature of the algorithm, the topmost $n+2$ elements of the stack[1] are

$$[B \to \beta_1 \bullet A\beta_2][A \to \bullet X_1 \ldots X_n] \ldots [A \to X_1 \ldots X_n \bullet]$$

The topmost $n+1$ of these are popped. A is pre-appended to the input. Finally, a shift step is performed.

Nondeterminism only occurs in two cases.

- If the current state is of the form $[A \to \alpha_1 \bullet B\alpha_2]$ and there is more than one rule with B at its lhs, different production steps are possible. This is called a *produce-produce conflict*.
- If the current state is of the form $[A \to \alpha_1 \bullet B\alpha_2]$, both a shift step and at least one production step is possible, provided the remaining input is headed by B. This is called a *produce-shift conflict*.

3.4 $SKLR(0)$ parsers

For most CFGs the $LL(0)$ parser is very nondeterministic. As soon as a nonterminal occurs as the lhs of more than one production, nondeterminism

Uniting states is a technique, taken from LR parsing, for increasing determinism. The transformation from LL to LR parsing can be seen to consist of two subsequent steps, which we will call *production step elimination* and *determinization*.

A glance at the $LL(0)$ parser shows that nondeterminism occurs at production steps. The first transformation step, production step elimination, consists in eliminating explicit production steps by uniting those dotted rules into one state that are interrelated by such a step.

Hence, in *singleton-kernel* $LR(0)$ *parsing*, or $SKLR(0)$ parsing, states are sets of dotted rules. Every state contains one proper dotted rule, its *kernel*. The other dotted rules in a state are those that can be obtained from the kernel by means of one or more production steps.

For an $SKLR(0)$ parser, being in some state implies that the parser is, in the virtual parse tree, *simultaneously* at all positions that are indicated by a sequence of dotted rules in this state, that starts with the kernel and of which every following dotted rule can be reached from the preceding one by means of a production step.

Let us discuss some differences with $LL(0)$ parsers.

- First of all, there are no production steps here. This is the very essence of our first transformation step. A sequence of consecutive production steps is made at once upon entering a new current state.
- Therefore, states changed from being dotted rules to sets of dotted rules.
- A reduction step involves popping n elements from the stack, rather than $n+1$, because a production step does not cause a push any more.

Nondeterminism occurs in an $SKLR(0)$ parser in the following cases.

- If some state contains two different dotted rules of the form $[A \to \alpha_1 \bullet X\alpha_2]$ and $[B \to \beta_1 \bullet X\beta_2]$, a shift step leaves the next current state not uniquely determined. This is a *shift-shift conflict*,
- If some state contains a dotted rule of the form $[A \to \alpha \bullet X\beta]$ and a dotted rule of the form $[A \to \alpha\bullet]$, so that both a shift and a reduction step are possible (a *shift-reduce conflict*),
- If some state contains more than one dotted rule of the form $[A \to \alpha\bullet]$, so that more than one different reduction step is possible (a *reduce-reduce conflict*).

As opposed to the $LL(0)$ parser, the $SKLR(0)$ parser applies to at least some left-recursive grammars. Though nondeterministic, $SKLR(0)$ parsers are terminating for a reasonable class of left-recursive CFGs. An advantage of $SKLR(0)$ parsers over $LR(0)$ parsers is that the parsing tables are more space efficient, in the worst case. Any $SKLR(0)$ parser will have a number of states that equals the sum of the lengths of the rhss. However, $SKLR(0)$ parsers are more nondeterministic than $LR(0)$ parsers.

[1] When writing down stack contents as a string, the top will be at this string's right.

3.5 $LR(0)$ parsers

As mentioned, three kinds of nondeterminism occur in $SKLR(0)$ parsers. The second transformation step, determinization, completely abolishes the shift-shift conflicts. Wherever such a conflict may occur, all possible next current states are taken together to form a new state. So, states in $LR(0)$ parsers will be unions of $SKLR(0)$ parser states. Unfortunately, shift-reduce and reduce-reduce conflicts remain, and generally even grow in number.

Hence, in $LR(0)$ parsers,

- states are unions of $SKLR(0)$ states, and
- there are no shift-shift conflicts.

Nondeterminism occurs in $LR(0)$ parsers only as shift-reduce or reduce-reduce conflicts.

4 Extending parsers with attribute evaluation

In this section, we add attribute evaluation to the $LL(0)$ parser and present transformations of this attributed parser, via an attributed $SKLR(0)$ parser to an attributed $LR(0)$ parser. It appears that production step elimination causes problems related to attribute evaluation. Remember that the discussion is restricted to ULAGs.

4.1 L-parser/evaluators

This subsection presents L-parser/evaluators. They form the basis for the parser/evaluators of this section.

When attribute evaluation is performed during parsing and the parse tree should not be stored, entirely nor partially, the attribute values should be calculated at the moment that the parser, during its walk through the virtual parse tree, is at the corresponding node. The attribute values are kept within the states on the stack.

First of all, we must capture the fact that the lexical analyzer yields, instead of a string of alphabet symbols, a string of pairs, each pair consisting of a symbol and a synthesized attribute value associated with the symbol. This also enables us to pre-append the lhs of a production, after a reduction step, to the remaining input, together with its synthesized attribute value.

Also, we have to be careful with the definition of the states of a parser/evaluator. In some way, the attribute values must be attached to the states. However, at parser/evaluator-generation time, attribute values are not known yet. They are calculated at run time. Therefore, we make a distinction between *static* and *dynamic* states. Dynamic states are those entities that occur on the stack during an actual run of the parser/evaluator. Static states underlie dynamic states. Static states are constructed at parser/evaluator-generation time. Obviously, the only part of a dynamic state that cannot be calculated at parser/evaluator-generation time concerns the actual attribute values.

4.2 $LL(0)$ parser/evaluators

This subsection introduces the $LL(0)$ parser/evaluator.

Within states of the form $[A \to \alpha_1 \bullet X\alpha_2]$, the value of the inherited attribute occurrence of X is kept. Within states of the form $[A \to \alpha_1 X \bullet \alpha_2]$, the value of the synthesized attribute occurrence of X is kept. Notice that in most states, being those that have at least one symbol (say Y) following and one symbol (say X) preceding the dot, carry a value of the inherited attribute occurrence of Y as well as the value of the synthesized one of X. Yet, the synthesized attribute occurrence of X is evaluated earlier than the inherited one of Y.

Now, let us consider the three kinds of steps of the $LL(0)$ parser.

- Consider the production steps. A new state of the form $[B \to \bullet\beta]$ is pushed. If $\beta \neq \varepsilon$, the inherited attribute value of the first symbol of β, say X, must be calculated. By the L-attributedness of the AG, it can only depend on the inherited attribute occurrence of B, of which the value can be found in the state $[A \to \alpha_1 \bullet B\alpha_2]$, below the current one on the stack.
- Consider the shift steps. A new state of the form $[A \to X_1 \ldots X_n X \bullet \alpha]$ is pushed. We have two cases.
  - If $\alpha \neq \varepsilon$, this state must contain the synthesized attribute value of X and the inherited attribute value of

the first symbol of α, say Y. The synthesized attribute value of X is copied from the remaining input. It has been put there by the lexical analyzer or by the preceding reduction action. The inherited attribute value of Y must be calculated. Because of the L-attributedness of the AG, it only depends on the synthesized attribute occurrences of $X_1, \ldots, X_n, X$ and on the inherited attribute occurrence of A. The synthesized attribute value of X is taken from the stack. The others can be found at a distance from the stack's top that equals the position number of their corresponding symbol in the string $X_n \ldots X_1 A$.

- If $\alpha = \varepsilon$, it suffices to copy the synthesized attribute value from the remaining input.

- Consider a reduction step with $A \rightarrow X_1 \ldots X_n$ as the production involved. No new states are pushed. However, A is pre-appended to the remaining input. It must be accompanied by its synthesized attribute value. This value is calculated by using the values of the synthesized attribute occurrences of $X_1, \ldots, X_n$ and the inherited attribute of A. These can be found at a distance from the stack's top that equals the position number of their corresponding symbol in the string $X_n \ldots X_1 A$. Also, a reduction step is only made if the semantic condition associated with the production evaluates to TRUE.

A dynamic state must contain an inherited and a synthesized attribute value. The function needed to calculate the inherited attribute value from another one is determined by the dotted rule itself. This is not the case for the synthesized attribute since its corresponding function is determined by the production applied beneath the dotted rule's production. Now, suppose we have the dotted rule $[A \rightarrow X_1 \ldots X_m \bullet Y_1 \ldots Y_n]$ as the state's core. If $m > 0$, the synthesized attribute value of X_m is simply copied into the synthesized attribute 'slot' of the state. If $n > 0$, we must calculate the inherited attribute value of Y_1, which depends on $m + 1$ other values, being the inherited attribute value of A and the m synthesized ones of $X_1, \ldots, X_m$. If $n = 0$, we do not have to calculate an inherited attribute value.

A static state in our $LL(0)$ parser/evaluator is a dotted rule, extended with the function that calculates the inherited attribute of this dotted rule. This is called an *extended dotted rule*. An extended dotted rule contains a *string* of functions[2] instead of just one. This is to prepare for future transformations. Extended dotted rules are denoted by $[A \rightarrow \alpha \bullet \beta | \varphi]$.

A static attributed $LL(0)$ state is an extended dotted rule. A dynamic attributed $LL(0)$ state is a triple consisting of the associated static state, and the associated synthesized and inherited attribute value, respectively.

An $LL(0)$ parser/evaluator is as (non)deterministic as its underlying $LL(0)$ parser.

4.3 $SKLR(0)$ parser/evaluators

This subsection will present $SKLR(0)$ parser/evaluators and the problems involved in their definition. Basically, two main techniques are used. First, we use string rewrite systems in order to capture semantic knowledge of the semantic functions. We need this knowledge in order to keep our states finite, but they also help decreasing nondeterminism. Second, the problem of how to address the proper inherited attribute value in a state is solved by including a kind of place marker in the dynamic states.

4.3.1 Production step elimination

Let us first consider which attribute values are associated with an $SKLR(0)$ state. First of all, we have the inherited and synthesized attribute value of the kernel of the state. Also, we have the inherited attributes of the other dotted rules. They have no synthesized attributes because they are starting dotted rules. So, every $SKLR(0)$ state contains exactly one synthesized attribute value, but generally more than one inherited attribute value. Hence, in order to extract an inherited attribute value from a state, one needs more arguments than just the state itself. A first guess, of course, is that the additional argument should be

[2] Concatenation is this string denotes function composition.

the dotted rule with which the intended value is associated. Unfortunately, this has two problems.

- Inherited attribute values (of a nonterminal A) are needed only when a shift or reduction-shift step is made according to a production that has A as its lhs. Therefore, at the moment of extracting the value, the only information available is the symbol immediately following the dot in the intended dotted rule (being A), and not the entire dotted rule. In general, the same symbol may immediately follow the dot in more than one dotted rule in one state.

- Generally, even the entire dotted rule is not sufficient to uniquely identify an inherited attribute value. More than one inherited attribute value may be associated with the same dotted rule in the same state.

In an elegant definition, a static attributed $SKLR(0)$ state would be a set of extended dotted rules, being the closure, with respect to some relation, of some kernel element. A first concern is whether the function, kept within the extended dotted rule, should denote the function needed to calculate the associated inherited attribute value from the one immediately preceding it (with respect to that relation), or the function needed to calculate the associated inherited attribute value from the inherited attribute value of the kernel. In this last case, we have a string of functions, which denotes composed functions, inside the extended dotted rule. We have decided for the second alternative.

Unfortunately, such a definition yields static states with infinitely many extended dotted rules in case of left-recursion. A partial solution to this problem may be obtained as follows.

In most of attribute grammar theory, the semantics of the semantic functions is not taken into account. Only attribute *dependencies* are important. Evaluator implementation is based on the a priori availability of implementations of the semantic functions. If we persist in this principle here, we cannot but forbid ULAGs with left-recursive underlying CFGs. However, being able to handle left-recursive grammars is a major advantage of LR parsing over LL parsing. It is not desirable to let the addition of simultaneous attribute evaluation nullify this. So, we are forced to pay attention to the semantics of the semantic functions.

There are many formalisms by which we might do this. Without indulging to elaborate discussions of this area of computer science, we present one solution, which enables to handle at least the most practical cases. This solution uses string rewrite systems.

4.3.2 String rewrite systems

A string rewrite system is a special kind of *rewrite system*. A (general) rewrite system contains an arbitrary set of *objects* and a binary relation $\Rightarrow$ on this set. A rewrite step transforms an object into another one according to a rewrite rule. An object that can be obtained from another object by a (possibly empty) sequence of rewrite steps, respectively one rewrite step, is called a *descendant*, respectively a *direct descendant*, of that object. An object is called *irreducible* if no rewrite rule can be applied to it any more. If an object can be rewritten into an irreducible object, this irreducible object is called a *normal form* of the original one. If no infinite sequences of rewrite steps can occur, the system is called *terminating*. The system is called *confluent* if any two different descendants of the same object have a common descendant. The system is called *complete* if it is both terminating and confluent. Complete rewrite systems implement a total function yielding the (unique) normal form for any given object.

A *string rewrite system* (or *semi-Thue system*, or *STS*, shortly) is a special kind of rewrite system. The objects are strings over some set of symbols. A rewrite rule here is a pair of strings. An application of such a rewrite rule to a string consists in substituting a substring, that matches the lhs of the rule, by the rhs of the rule. A *finite* STS has finitely many string rewrite rules.

Now, STSs are used in the following way. When constructing a static state, which consists of extended dotted rules, the function strings in these extended dotted rules are first subjected to the STS. Using STSs is only a partial solution to the problems mentioned: particular ULAGs will still have to be rejected, because they yield infinite static states even after the use of the STS.

4.3.3 High-lights

As mentioned earlier, we also need to include, in our dynamic states, an indication that allows us to extract the proper inherited attribute value from it, the next time such a value is requested. For sake of brevity, the complicated discussion of these high-lights is omitted. We restrict ourselves to mentioning that, in principle, extended dotted rules suffice as high-lights. However, more sophistication, that is, using so-called *extended symbols* or even *extended sets* as high-lights decreases the amount of nondeterminism. Also, high-lights appear to have syntactical consequences: a particular high-light in a dynamic state restricts the set of possible steps. For instance, a certain high-light may forbid a shift of some symbol a, even when the underlying static state allows such a shift step.

Unlike $LL(0)$ parser/evaluators, $SKLR(0)$ parser/evaluators may show more nondeterminism than their underlying parser. One can distinguish *syntactic conflicts* form *semantic conflicts*. Semantic conflicts may occur when high-lights are changed and the next high-light is not uniquely determined.

4.4 $LR(0)$ parser/evaluators

States in $LR(0)$ parsers are unions of $SKLR(0)$ parser states. Analogously, static attributed $LR(0)$ states are unions of static attributed $SKLR(0)$ states. This causes a technical problem, because the static states may now contain multiple kernel elements. The solution involves the introduction of so-called *selection functions*. These form the most important difference between $LR(0)$ and $SKLR(0)$ parser/evaluators. However, we will refrain from discussing this problem here.

4.5 Examples of practical string rewrite systems

In this subsection, we present some examples of STSs that are applicable to frequently appearing semantic functions in attribute grammars.

First of all, copy rules often occur. A copy rule occurs when an attribute evaluation function is the identity function. For the identity function, say I, we have a simple rule.

$$\mathrm{I} \to \varepsilon$$

In other words, the identity function can simply be removed from any string of functions.

A second class of functions suitable for rewriting is formed by the constant functions. Let $c : A \to B$ be any constant function and $f : B \to B$ any function. Then, the rule

$$cf \to c$$

can be added. It expresses that any function applied before the constant function is superfluous.

Finally, we notice that another rule can be used when f is, for instance, the logical negation, or a function, taking and yielding pairs, that exchanges the pair's first and second constituent. This rule is

$$ff \to I$$

5 Implementing parsers

In this section, we discuss implementation issues of the nondeterministic parsers as presented in Section 3. A main problem here is that the conceptual nondeterminism must be implemented on a deterministic machine. A first remark is that all parsers discussed here can be made table-driven in the usual way.

5.1 $LL(0)$ parsers

In $LL(0)$ parsers, a production step is performed as follows. Conceptually, the parser initiates a new parser for every possible new state to be pushed.

A shift step is performed as follows. Suppose the current state is $[A \to \alpha_1 \bullet X\alpha_2]$ and Y heads the remaining input. If $X = Y$, $[A \to \alpha_1 X \bullet \alpha_2]$ is pushed and Y is removed from the remaining input. If $X \neq Y$, no shift step can be performed.

A reduction step is performed as follows. If the current state is not of the form $[A \to X_1 \ldots X_n \bullet]$, no reduction step can be performed. If it is, $n+1$ elements are popped from the stack and A is preappended to the input. After that, a compulsory shift step follows.

If no production step, as well as no shift nor reduction step is possible, the parser dies. If more than one kind of step is possible, new parsers are initiated for both alternatives. In $LL(0)$ parsers

this can occur with production and shift steps (a produce-shift conflict). If a reduction-shift step is possible, no other steps are.

5.2 *SKLR*(0) parsers

In *SKLR*(0) parsers, a shift step is performed as follows. Suppose X heads the remaining input. Then, the set of all dotted rules in the current state of the form $[A \rightarrow \alpha_1 \bullet X\alpha_2]$ is determined. For every such dotted rule, a new parser is initiated, which pushes the state with $[A \rightarrow \alpha_1 X \bullet \alpha_2]$ as its kernel.

A reduction step is performed as follows. The set of all dotted rules of the form $[A \rightarrow X_1 \ldots X_n \bullet]$ in the current state is determined. For every such dotted rule, a new parser is initiated, which pops n elements from the stack.

If no shift nor reduction step is possible, the parser dies. If both shift and reduction steps are possible, new parsers are started for both.

5.3 *LR*(0) parsers

In *LR*(0) parsers, a shift step is performed as follows. Suppose X heads the remaining input. Then, the set of all dotted rules in the current state of the form $[A \rightarrow \alpha_1 \bullet X\alpha_2]$ is determined. If this set is empty, the parser dies. If not, a new state is pushed with this set as its kernel set.

Reduction steps are performed as in *SKLR*(0) parsers. If no shift nor reduction step is possible, the parser dies. If both shift and reduction steps are possible, new parsers are started for both.

5.4 Handling nondeterminism: a simple approach

Our parsers contain a possibly huge amount of nondeterminism. Because our strategy is to try out all possibilities in case of nondeterminism, we must have a way to implement this. A simple approach is the following: a parser first determines which production, shift and reduction steps are possible. Suppose there are n different possible next steps. Then the parser copies its instantaneous description n times, yielding one copy for every possible step. Then, for every copy, it performs the associated step and initiates a new parser to continue parsing with the resulting new instantaneous description. After that, it dies. If $n = 0$, the parser dies without initiating new ones.

A parser stops (successfully) when its instantaneous description has the final state q_F as its current state.

For the time being, a sequential implementation is intended. Words like 'parallelism', 'synchronization', and different 'parsers' are used in a conceptual sense.

5.5 Sharing instantaneous descriptions

There is a lot of unnecessary copying of instantaneous descriptions in the simple approach. Major portions of the copies will coincide. Sharing these coinciding parts may substantially increase efficiency.

A parser can be seen to move on a tape that is formed by the stack and the remaining input. Upon a nondeterministic choice, originally, the tape was copied several times before the step. However, major parts of the tape are the same for all copies. These parts, a bottom part (prefix) of the stack and a suffix of the remaining input can be shared by all alternatives. This yields a data structure which we will call a *graph-structured instantaneous description.*

Basically, this is a directed acyclic graph, with one source (the bottom of the stack) and one sink (the end of the input). It is maintained in such a way, that every path from the source to the sink corresponds to a single instantaneous description. On every such path, there is one current position, which marks the border between the stack and the input of the corresponding instantaneous description. Obviously, maximal sharing is reached when no parent has different but equally labeled children (at the stack side) and no child has different but equally labeled parents (at the input side). This technique will save space, and may cost time (because of the additional problems of manipulating shared data), but it may save much more (because instantaneous descriptions will not be exhaustively copied).

Though this graph-structured instantaneous description may seem symmetric, there is an asymmetry to be found in that the stack part of an instantaneous description is both growing and shrinking during a run, whereas the input part only shrinks (after initialization). Therefore, re-

maining inputs of two different parsers will always be such, that one is a suffix of another. So, at the remaining-input side, it is very easily assured that no parser destroys other parsers' data. This is different at the stack side. There, a reduction step, causing stack elements to be popped, endangers other parsers' data. In this case, the instantaneous description should be partially copied (that is, the endangered part only) before the step is actually performed. The partial copy is connected with the original at the border of the endangered part.

There is no reason, other than for efficiency, to decrease the amount of sharing used. One might even refrain from sharing data at one of both sides of the shared instantaneous description. In this case, our shared data structure has the structure of a tree. Two possibilities occur. The first is when the input side lacks all sharing. This yields a *tree-structured instantaneous description*, which has one source (the bottom of the stack) and many sinks, indicating the end of the remaining input for every separate parser. The other possibility occurs when the stack side lacks all sharing. This causes a data structure with many sources and one sink. We will call this a *funnel-structured instantaneous description*. We refrain from giving technical details.

5.6 Synchronizing the parsers

Additional efficiency may be gained by requiring parallelism to be synchronous, or, even stronger: by demanding synchronization on shift steps.

To see this, notice that the shift step is the only step in which the input is affected (because of the combined reduce-shift step). Then, if all parsers remove the same symbol from the input at the same time, they all will always have the same remaining input, which therefore can be factored out of the graph-structured instantaneous description. This technique will save space, but cost time, since parsers may have to wait for synchronization.

Because the remaining input is no longer part of the graph-structured instantaneous description, it is rather a *tree-structured stack*, which has one source and multiple sinks, one for the top of the stack of every single parser. Now, because the steps of the parser depend on the current state only, we know that all branches of this tree-structured stack, that have the same state at their leaves, will have identical tree-structured substacks originating from them in the future. Therefore, it might be advantageous to unify these leaves into one. This will save space, as well as time and/or processors, simply because identical subruns will be performed only once. However, it may also cost time, because searching for identically labeled leaves takes time. This data structure is called a *graph-structured stack*. It has one source and, at any moment, generally multiple sinks, one for every state occurring at the top of a stack at that moment.

The idea of a graph-structured stack is not new. It originated from Tomita (1985), in which, however, it is only used in the context of generalized *LR* parsing. However, graph-structured stacks were introduced differently. There is no notion of asynchronous (pseudo-)parallelism nor of graph-structured instantaneous descriptions. Moreover, the concept of a tree-structured stack is different, in that the sharing is the other way around: the tree has the current state at its root, and the stack bottom (or rather bottoms) at its leaves. We automatically arrived at the reverse notion of tree-structured stacks by not presupposing shift synchronization. In order to stay consistent with earlier terms, we use the name *funnel-structured stack* for Tomita's notion of tree-structured stacks.

5.7 Tree- and graph-structured stacks in detail

A detailed discussion of the use of tree- and graph-structured stacks in generalized $LL(0)$, $SKLR(0)$, and $LR(0)$ parsing is given in Oude Luttighuis and Sikkel (1992), but omitted here for brevity's sake.

5.8 Parallelism in parsers

Here, we will discuss ways to actually implement the defined parsers and parser/evaluators in parallel.

The parallelizations of these algorithms can be classified according to which data structure(s) is (are) distributed over different processors. By this distribution, we do not necessarily mean physical distribution, but rather conceptual distribution. Of course, parallelism may also be ob-

tained by distributing control structure, but in our algorithms, control structure is captured in a data structure.

The only data structure maintained in our parsers and parser/evaluators is the (possibly tree- or graph-structured) instantaneous description. Yet, it consists of the (remaining) input and the stack part.

Because activity only occurs at the top of the stack and the head of the input, it seems unprofitable to cut them into subsequent pieces and assign these to processors. So, the only fruitful parallelism may be found in the pseudo-parallelism already present. Different processors simply process the different alternatives originating from nondeterministic decision points.

5.8.1 Dividing multiple linear stacks and tree-structured stacks

Parallel implementation of Tomita's algorithm can be found in (Tanaka and Numazaki, 1989; Numazaki and Tanaka, 1990). Both divide the different alternatives occurring at nondeterministic decision points over the processors. The first uses no sharing whatsoever of the instantaneous description: several different copies of (linear) stacks are processed by different processors. The second one uses the tree-structured stack.

5.8.2 Dividing graph-structured stacks

Graph-structured stacks were introduced, because tree-structured stacks show many identical activities, because many top nodes may be labeled with the same state. As discussed earlier, sharing is best applied immediately after the shift steps that start a tst-step.

What we might do is take a processor for every state in the parser[3]. Its task is to perform one tst-step to the (shared) wait node that is labeled with its associated state.

Because all shift steps shift the same symbol, the maximum number of wait nodes resulting is the maximum number of states that can occur after a shift of a particular symbol. So, if we choose to assign a processor to all parser states, many of them will be inactive at some time. This can be improved by partitioning the parser states into a set of blocks. Every block corresponds to a grammar symbol X and contains those states that can become current state after a shift of X. Now, we take a number of processes such that, to each of them, (at most) one state in every block is assigned. This calls for a number of processors that equals the maximum block size.

5.8.3 Dividing the input

There exist other parallelizations of Tomita's algorithm. In (Lankhorst and Sikkel, 1991) the PBT (Parallel Bottom-up Tomita) algorithm is presented. In this algorithm, the input is divided over processors such that every processor processes one input symbol. The processors operate in a pipeline, communicating *marked symbols* from the end to the beginning of the string. A marked symbol consists of a grammar symbol (terminal or nonterminal) and two numbers, which indicate the left and right border of a substring of the input string, that is derivable from that grammar symbol. Actual recognition of a new marked symbol with left border i is done by the processor associated with the i^{th} input symbol.

It was stated in the beginning of this section that it seems unprofitable to cut the input (or the stack) into subsequent pieces and assign these to processors, because activity only occurs at the head of the input (and the top of the stack). Yet, this holds only for true parallelizations of Tomita's algorithm. PBT is rather another algorithm than Tomita's. The facts that PBT requires different table construction and different manipulation of the graph-structured stack support this view. Also, whereas Tomita's algorithm cannot handle hidden-left-recursive CFGs, PBT can.

6 Implementing parser/evaluators

This section discusses implementation issues of the nondeterministic parser/evaluators as presented in section 4. The underlying syntactic part

[3]That is, a processor is associated with every parser state, not with every instance of a state on the graph-structured stack.

can be made table-driven in the usual way. Additionally, in $SKLR(0)$ and $LR(0)$ parser/evaluators, high-lights have to be taken into account as well in the parser tables.

6.1 Calculating attribute values

Attribute values are computed upon syntactical steps. The (candidate) steps to be taken are chosen on purely syntactical grounds. In reduction steps, the semantic condition may prohibit the step. In other cases, the calculation of attribute values simply follows syntactical processing. The argument values are extracted from the other states on the stack (and copied from the remaining input, in case of a shift step).

6.2 Handling nondeterminism

In the parser/evaluators presented, attribute evaluation can influence parsing in two ways. First, a reduction step will be prevented if the semantic condition evaluates to `FALSE`. Second, the high-lights have syntactical consequences. Yet, the techniques of handling nondeterminism, as mentioned for parsers, can in principle all be used for parser/evaluators as well. However, implementing a graph-structured stack will not be as profitable for parser/evaluators as it is for parsers. This has two reasons.

- As opposed to the steps of the parsers, the steps of the parser/evaluator are not completely determined by the current state only. For the evaluation of attribute values, argument values are needed that reside in states further down in the stack. This causes complications when, somewhere within the sequence of states from which attribute values must be extracted, sharing has been applied, so that a new kind of semantic nondeterminism occurs. In other words, because more than one stack is associated with a top node in graph-structured stacks, different attribute values may be extracted from states deeper in the stack.

- States may only be shared if they are identical. Since dynamic states now include attribute values, these have to be identical as well. It is doubtful whether the time saved by sharing identical states outweighs the time lost in verifying that states are identical, given the dynamic nature of attribute values.

Both tree- as well as graph-structured stacks suffer from the fact that a larger portion of the stack is inspected for each step. This may cause more read conflicts in a parallel implementation. Still, a tree-structured stack seems feasible.

6.3 Parallel parser/evaluators

This section will discuss ways to implement the defined parser/evaluators in parallel.

Because parsers are the skeleton of our parser/evaluators, we review the opportunities for parallelism, which are present in our parsers, and discuss whether they can be applied to parser/evaluators as well.

Division of multiple linear stacks and tree- and graph-structured stacks can be carried over to parser/evaluators. However, there are two difficulties.

- Graph-structured stacks are hardly useful for parsing/evaluation (as discussed earlier). Therefore, the parallel parsing technique for dividing the graph-structured stack over a (statically bounded) number of processors is not applicable.

- In tree-structured stacks (as well in graph-structured stacks) the sharing of parts of the data structure introduces the possibility of read conflicts.

Unfortunately, dividing the input seems unsuitable as well, because the L-attributedness of ULAGs imposes a very sequential nature on attribute dependencies. Only when a left-to-right dependency is absent, possibilities for parallelism appear. This occurs, for instance, in case there are no inherited attributes, but only synthesized ones. However, if we allow ourselves to use dynamic evaluation techniques, this changes, because we may postpone evaluation. This is the subject of the next subsection.

6.4 Dynamic attribute evaluation

Although dynamic attribute evaluation is not a main topic of this paper, we will spend some remarks on it.

Dynamic attribute evaluation abandons the need to (completely) evaluate attribute values before they are used. Generally, a directed graph is maintained in which nodes are labeled with semantic functions and have outgoing arcs to all their arguments. Let us call such a graph an *attribute graph*. This graph can essentially be handled by two techniques.

- The first technique calculates an attribute value in the attribute graph only when all its argument values are completely evaluated. Let us call this the *evaluation-before-use* technique.
- In the other technique, the attribute graph is viewed as a syntactic description of a value. Rewrite rules are applied to it whenever this is possible. A rewrite rule may, in principle, be performed to any part of the graph. This is the *graph-rewriting* technique.

In both techniques, the attribute graph grows whenever the parser enters a new state and shrinks whenever a new attribute value is calculated, or a rewrite rule has been applied.

Dynamic evaluation may be helpful in parsing/evaluation in two ways. First, it allows for postponing (complete) evaluation of attribute values. Second, related with that, it allows for a less rigid approach to infinity problems in case of left recursion. Apparently, a combination of generalized parsing/evaluation techniques and dynamic evaluation may be desirable.

The fact that dynamic evaluation partly frees attribute evaluation from problems caused by syntactical processing also offers new possibilities for parallelism.

For a more elaborate discussion, see Oude Luttighuis and Sikkel (1992).

7 Related approaches

The only GLR parser generator that we know of is the incremental parser generator IPG (Heering et al., 1990; Rekers, 1992), from the ESPRIT project GIPE (generation of interactive programming environments). They chose Tomita's algorithm as a starting point for their incremental parser generation because it provides a good mix of generality and efficiency. One of the reasons for accepting arbitrary context-free languages is that the grammar is allowed to be *modular*, and none of the classes of LR-grammars is closed under composition. Rekers' thesis does not discuss attribute evaluation in the incremental parser generator.

Affix Grammars over a Finite Lattice (AGFLs) (Weijers, 1986) are a sub-class of affix grammars specifically designed for natural languages. The formalism can be located somewhere between attributed grammars and feature-structure grammars (Shieber, 1986). There are two basic ways to parse an AGFL: (1) Evaluate the affix values on-the-fly during the construction of the parse forest, or (2) compute a parse forest according to the underlying context-free grammar and decorate it with affix values afterwards. Koster (1991) claims that the first approach is more practical, provided that his "Recursive Backup" algorithm (Koster, 1975) is used, rather than an Earley or Tomita parser (despite the exponential worst-case complexity of the Recursive Backup algorithm). Nederhof and Sarbo (1993) claim the reverse, however. They discuss how ambiguity can be handled practically in an interactive environment.

A generalization of an LC parser, based on the recursive backup method mentioned above, is used for the closely related formalism of Extended Affix Grammars in (Meijer, 1986).

A generalized LR parser for a query language for logical databases is described in (Lang, 1988).

8 Conclusions

Contributions of this paper are

- a systematic treatment of generalized *LL* and *LR* parsing,
- the description of *SKLR* parsing as an intermediate form of these,
- a correspondingly systematic treatment of attribute evaluation during *LL*, *SKLR*, and *LR* parsing, with a precise identification of problematic issues in the case of *SKLR* and *LR* parsing,
- (partial) solutions to these problems, viz. high-lights, string rewrite systems, and selection functions,

- the technique of attribute evaluation during generalized *SKLR* parsing, which is to be preferred to evaluation during generalized *LR* parsing, because it avoids some technical complications while nondeterministism is only marginally increased.

Future work may be done on the following problems.

- What can be gained by adding the use of look-ahead information to *SKLR* parsing?
- What is the relationship between *SKLR* and *LC* parsing?
- Can our evaluation techniques be efficiently combined with dynamic evaluation techniques?
- Is the use of string rewrite systems a real gain in practice?

References

Heering, J. — P. Klint — J. Rekers (1990). Incremental generation of parsers. *IEEE Transactions on Software Engineering*, SE–16:1344–1351.

Knuth, D.E. (1965). On the translation of languages from left to right. *Information and Control*, 8:607–639.

Koster, C.H.A. (1975). A technique for parsing ambiguous grammars. In D. Siefkes, editor, *GI — 4. Jahrestagung.* Lecture Notes in Computer Science 26.

Koster, C.H.A. (1991). Affix grammars for natural languages. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems (Prague, Czechoslovakia, June 4–13, 1991)*, pages 469–484, Berlin, Germany. Springer-Verlag. Lecture Notes in Computer Science 545.

Lang, B. (1988). Datalog automata. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness (Jerusalem, Israel, 1988)*, pages 389–404.

Lankhorst, M. — K. Sikkel (1991). PBT: A parallel bottom-up Tomita parser. Memoranda Informatica INF 91–69, Department of Computer Science, University of Twente, Enschede, The Netherlands, September.

Meijer, H. (1986). *Programmar: A Translator Generator.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands.

Nederhof, M.-J. — J.J. Sarbo (1993). Efficient decoration of parse forests. In H. Trost, editor, *Feature formalisms and linguistic ambiguities*, pages 95–109, Chicester, U.K. Ellis Horwood.

Numazaki, H. — H. Tanaka (1990). A new parallel algorithm for generalized LR parsing. In *Proceedings of the 13th International Conference on Computational Linguistics (Helsinki, Finland, 1990) (Vol. 2)*, pages 304–310.

Oude Luttighuis, P. — K. Sikkel (1992). Attribute evaluation during generalized parsing. Memoranda Informatica 92–85, Department of Computer Science, Enschede, The Netherlands.

Rekers, J. (1992). *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands.

Shieber, S.M. (1986). An introduction to unification-based approaches to grammar. CSLI Lecture Notes 4, Center for the Study of Language and Information, Stanford University, Stanford, California, USA.

Sippu, S. — E. Soisalon-Soininen (1990). *Parsing Theory*, volume II LR(k) and LL(k) Parsing. Springer-Verlag, Berlin, Germany.

Tanaka, H. — H. Numazaki (1989). Generalized LR parsing based on logic programming. In *Proceedings of the International Workshop on Parsing Technologies (Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1989)*, pages 329–328.

Tomita, M. (1985). *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, Massachusetts, USA.

Weijers, G.A.H. (1986). Affix grammars over finite lattices. Report No. 94, Department of Computer Science, University of Nijmegen, Nijmegen, The Netherlands.

A Proof-Theoretic Reconstruction of HPSG*

Stephan Raaijmakers

Institute for Language Technology and Artificial Intelligence ITK
P.O.Box 90153, 5000 LE Tilburg, The Netherlands
email: stephan@kub.nl

Abstract

A reinterpretation of Head-Driven Phrase Structure Grammar (HPSG) in a proof-theoretic context is presented. This approach yields a *decision procedure* which can be used to establish whether certain strings are generated by a given HPSG grammar. It is possible to view HPSG as a fragment of linear logic (Girard, 1987), subject to partiality and side conditions on inference rules. This relates HPSG to several categorial logics (Morrill, 1990). Specifically, HPSG signs are mapped onto quantified formulae, which can be interpreted as *second-order* types given the Curry-Howard isomorphism. The logic behind type inference will, aside from the usual quantifier introduction and elimination rules, consist of a partial logic for the undirected implication connective.

It will be shown how this logical perspective can be turned into a parsing perspective.

The enterprise takes the standard HPSG of Pollard — Sag (1987) as a starting point, since this version of HPSG is well-documented and has been around long enough to have displayed both merits and shortcomings; the approach is directly applicable to more recent versions of HPSG, however. In order to make the proof-theoretic recasting smooth, standard HPSG is reformulated in a binary format.

1 Introduction

The main concern of this paper lies in building a parser for HPSG. The result of the enterprise should meet the following desiderata:

- The parser should interpret the original grammatical theory, or as close a dialect as possible.
- The parser should separate grammatical theory from parsing issues.
- The parser should make an operationalisation of the grammatical theory explicit, as declaratively as possible.
- It should be easy to alter the grammatical theory.
- The parser should have reasonable time/space complexity.

Existing parsers for HPSG do not obey these demands; e.g., the Popowich/Vogel parser (Popowich — Vogel, 1990) violates the second, fourth and fifth demand; the LiLog STUF environment (Dörre — Raasch, 1991) violates the first, third, and fifth. For a full comparison, see Raaijmakers (forthcoming).

In the *parsing-as-deduction* field, several parsing routines have arisen from proof-theoretic investigations (Moortgat, 1988; König, 1989). While these routines are not all among the most efficient, once a proof-theoretic formulation of HPSG has been made, one can benefit from these results.

*This research was carried out within the framework of the research programme 'Human-Computer Communication using natural language' (MMC). The MMC programme is sponsored by Senter, Digital Equipment B.V., SUN Microsystems Nederland B.V. and AND Software.

Some terminological remarks: we refer to HPSG of Pollard — Sag (1987) with '(classical) HPSG', and to its type-theoretic (deductive) equivalent with '$\mathcal{D}$-HPSG'.

2 An overview of HPSG

HPSG is a lexicalist, feature-based formalism for syntactic and semantic analysis of natural language. HPSG puts all relevant linguistic information in the lexicon, and has general rules and principles governing the construction of phrases from subphrases.

As a syntactic formalism, HPSG divides the labour of tree construction into separate processes of *mobile construction* and *mobile ordering*. A mobile is a tree-like structure with unordered trees; actually, a mobile can be interpreted as a description of a set of trees.

Socalled *immediate dominance* (ID) rules build these mobiles, which are then turned into trees by *linear precedence* (LP) principles. HPSG is a feature-based formalism, employing various feature mechanisms transporting feature information through feature structures. In HPSG, lexemes are bundles of so-called attribute-value pairs

$$\begin{bmatrix} A_i & V_i \\ \vdots & \vdots \\ A_n & V_n \end{bmatrix}$$

where A_j is a certain linguistic (phonological/syntactic/semantic) property taking its specification from a set of values containing V_j. These bundles are called *signs*. The reader is referred to Pollard — Sag (1987,1992) for a full overview of the various attributes and their values. The generic structure of main signs in HPSG is

$$\begin{bmatrix} \texttt{phon} & \dots \\ \texttt{syn} & \dots \\ \texttt{sem} & \dots \\ \texttt{dtrs} & \dots \end{bmatrix}$$

where the `phon`, `syn`, `sem` and `dtrs` values describe respectively the phonological, syntactic, semantic and configurational properties of the sign.

Attributes take either atomic or complex values; an attribute like `person` ranges over the set {`first`, `second`, `third`}, whereas an attribute like `dtrs` (describing daughters of phrases) takes full signs as values. The notion of *head* is a central concept in HPSG. Basically, a head of a phrase is a subphrase determining the relevant combinatorial properties of the sign. Heads can be phrasal or lexical; lexical heads are simply signs having no daughters. For instance, the head of a VP *sees Mary* is the verb *sees*; The grammatical properties of *sees* determine the properties (viz. agreement) of the VP as a whole, and not those of the direct object *Mary*. The head of the sentence *John sees Mary* is the VP *sees Mary*.

HPSG heavily leans on the notion of *unification* (Shieber, 1986). Simplifying matters somewhat, two signs S_1 and S_2 are unifiable with each other, written $S_1 \sqcup S_2$, if for any attribute they are both specified for, they bear non-conflicting values. Further, any fully disjunct parts of two signs (consisting of different attribute-value pairs) of the two signs can be combined directly. So,

$$\begin{bmatrix} \texttt{number} & \texttt{sg} \\ \texttt{person} & \texttt{first} \end{bmatrix} \sqcup \begin{bmatrix} \texttt{gender} & \texttt{fem} \\ \texttt{case} & \texttt{dative} \end{bmatrix} = \begin{bmatrix} \texttt{number} & \texttt{sg} \\ \texttt{person} & \texttt{first} \\ \texttt{gender} & \texttt{fem} \\ \texttt{case} & \texttt{dative} \end{bmatrix}$$

Likewise,

$$\begin{bmatrix} \texttt{syn}\,|\,\texttt{loc}\,|\,\texttt{head}\,|\,\texttt{maj} & \texttt{n} \\ \texttt{agr} & \begin{bmatrix} \texttt{gender} & \texttt{fem} \\ \texttt{number} & \texttt{sg} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \texttt{gender} & \texttt{fem} \\ \texttt{person} & 2 \end{bmatrix} = \begin{bmatrix} \texttt{syn}\,|\,\texttt{loc}\,|\,\texttt{head}\,|\,\texttt{maj} & \texttt{n} \\ \texttt{agr} & \begin{bmatrix} \texttt{gender} & \texttt{fem} \\ \texttt{number} & \texttt{sg} \\ \texttt{person} & 2 \end{bmatrix} \end{bmatrix}$$

But of course the following fails:

$$\begin{bmatrix} \texttt{number} & \texttt{sg} \\ \texttt{person} & \texttt{first} \end{bmatrix} \sqcup \begin{bmatrix} \texttt{number} & \texttt{plur} \end{bmatrix}$$

2.1 Immediate dominance (ID) rules

ID rules describe admissible dominance structures, which can be interpreted as mobiles: tree-like structures with unordered branches. The

rules themselves take the form of (partially specified) signs (just like HPSG's principles), applying as felicity constraints to signs to be combined.

Rule 1 (**R1**)

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \langle\rangle \\ \texttt{dtrs}\ \left[\begin{array}{l}\texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc} \mid \texttt{lex}\ - \\ \texttt{compdtrs}\ \langle \texttt{\_d}\rangle\end{array}\right]\end{array}\right]$$

Rule 2 (**R2**)

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \langle \texttt{\_s}\rangle \\ \texttt{dtrs}\ \left[\texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc}\left[\begin{array}{l}\texttt{head} \mid \texttt{inv}\ - \\ \texttt{lex}\ +\end{array}\right]\right]\end{array}\right]$$

Rule 3 (**R3**)

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \langle\rangle \\ \texttt{dtrs}\ \left[\texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc}\left[\begin{array}{l}\texttt{head} \mid \texttt{inv}\ + \\ \texttt{lex}\ +\end{array}\right]\right]\end{array}\right]$$

Rule 1 licenses signs having a non-lexical (i.e. phrasal) head daughter and being fully saturated, that is, signs having a subcat(egorisation) list of length zero (written as $\langle\rangle$). There is one complement daughter (indicated with the variable '_d'); an example would be an S having a VP as head daughter and a subject as only complement daughter. The `loc` attribute is used to describe "local" properties of a sign, such as lexicality and subcategorisation demands; this contrasts with the `bind` attribute, describing anaphoric links over signs.

Rule 2 caters for instance for VP's, which have a lexical head daughter (the verb), and are one short of becoming saturated: they subcategorise for a subject.

Rule 3 admits of saturated signs with a lexical, inverted head daughter, like in *Is John sleeping?*, the head daughter of which is the finite auxiliary *Is*, which subcategorises for both an infinitival VP and a nominative NP.

2.2 Linear Precedence (LP) principles

LP principles turn mobiles into genuine trees by imposing order on sister nodes.

Constituent Order Principle

$$\left[\begin{array}{l}\texttt{phon}\ \texttt{order} - \texttt{constituents}(\boxed{1}) \\ \texttt{dtrs}\ \boxed{1}\end{array}\right]$$

Linear Precedence Constraint 1 (**LP1**)
`head[lex +] < [ ]`

Linear Precedence Constraint 2 (**LP2**)
`complement << complement[lex -]`

The operation `order-constituents` gives the disjunction of all permutations of the phonology of the daughters. At least one of these permutations will have to be consistent with the constraints of order expressed by the LP principles, which are specific for English (and related languages):

LP1 says that lexical heads precede all their sisters (the empty sign [] acts like a "wildcard" symbol here, unifying with every sign). LP2 says that less oblique complements precede more oblique phrasal sisters; << is precedence between oblique elements, where obliqueness corresponds inversely to degree of obligatoriness. Subjects, for instance, are in Germanic languages less oblique than direct objects, which means they are more obligatory: they cannot be omitted, in general.

John eats.

*eats an apple.

The degree of obliqueness is mirrored (in reverse) by the order of complements on the `subcat` list of signs: the less oblique elements follow the more oblique elements.

2.3 Feature transport principles

Various principles take care of the distribution of feature information in a feature structure, defining the paths along which information percolates upwards. The concept of *reentrancy* expresses the sharing of information by several attributes across a sign, using boxed integers to identify attributes.

Head Feature Principle

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{head}\ \boxed{1} \\ \texttt{dtrs} \mid \texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc} \mid \texttt{head}\ \boxed{1}\end{array}\right]$$

Subcat(egorisation) Principle

$$\left[\begin{array}{l}\texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \boxed{2} \\ \texttt{dtrs}\ \left[\begin{array}{l}\texttt{headdtr} \mid \texttt{syn} \mid \texttt{loc} \mid \texttt{subcat}\ \boxed{1} + \boxed{2} \\ \texttt{compdtrs}\ \boxed{1}\end{array}\right]\end{array}\right]$$

(L1+L2 is the concatenation of the two lists L1 and L2.)

Semantics principle (simplified)

$$\left[\begin{array}{l}\textbf{sem}\ \left[\ \textbf{cont}\ s-c-s(\boxed{1},\boxed{2})\ \right] \\ \textbf{dtrs}\ \left[\begin{array}{l}\textbf{headdtr}\mid\textbf{sem}\mid\textbf{cont}\ \boxed{1} \\ \textbf{compdtrs}\mid\textbf{sem}\mid\textbf{cont}\ \boxed{2}\end{array}\right]\end{array}\right]$$

The HFP enforces identity between the head features of the head daughter and the mother sign.

The subcat principle decomposes the subcat list of a head daughter in two parts of arbitrary (non-empty) length: the first part should correspond to the value of `compdtrs`, the second part becomes the value of the `subcat` attribute of the mother sign.

The semantics principle states that the semantics of a mother sign must consist of the combination of the semantics of the head daughter and the complement daughters (the operation `successively-combine-semantics`, abbreviated as `s-c-s`, does just this.)

2.4 Sample derivation

The derivation in figure 1 shows the operation of the various principles and rules. Notice that HPSG as presented in Pollard — Sag (1987) assumes that nouns are heads selecting for determiners. Also, Pollard and Sag assume that heads themselves participate in the obliqueness hierarchy: they are more oblique than all their complements. Thus, LP2, once formulated as

`complement << complement[lex -]`

Pollard — Sag (1987:176) orders phrasal heads after their complements (for instance VP's after subjects). In order to derive the order 'the cat' rather than 'cat the', Pollard and Sag assume that the head noun 'cat' becomes phrasal by the fact that Rule 2 is applicable to it as a lexical sign (cf. op.cit. p.153); so, rule application turns `[lex +]` into `[lex -]`.

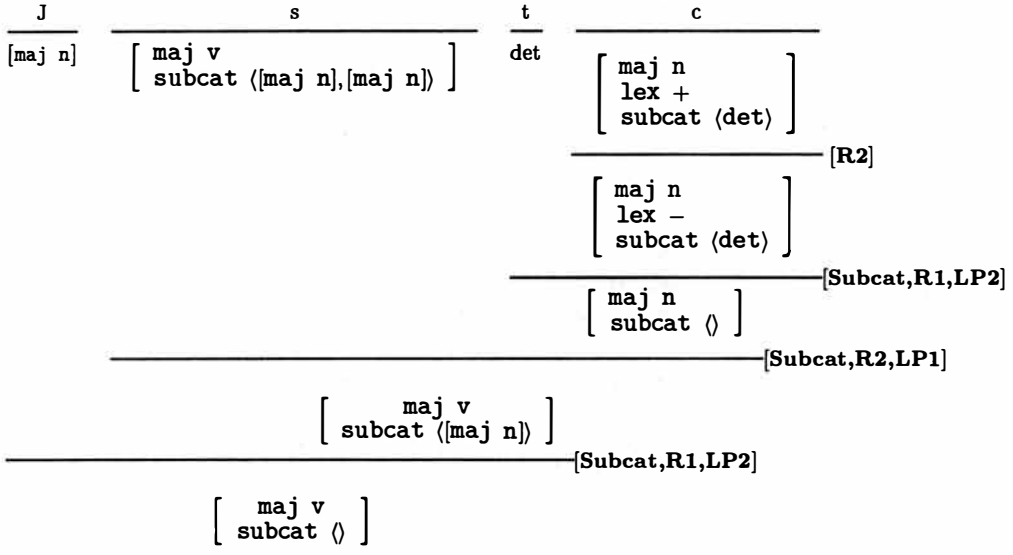


Figure 1: John sees the cat.

2.5 Tree arity

Classical HPSG is not strictly binary-branching: the Subcat principle allows for the combination of an *n*-ary functor with *n-1* arguments at once; e.g. the combination of a ditransitive verb taking three NPs (two complements and one subject) with two of its complements.

Classical HPSG is *non-monotonous*: as mentioned in the previous section, Rule 2 is able to change the lexicality of a sign by vacuously applying to that sign. This underdocumented feature of HPSG has several drawbacks, most significantly, the fact that the operation is not structure preserving: usually, signs evolve from subsigns under unification; here, a + value becomes a − value, which unification cannot possibly account

for. Conceptually, the lexicality feature seems to be *derivable*, since, only those signs are non-lexical (phrasal) which carry at least one daughter.

The HPSG theory as originally put forward by Pollard — Sag (1987) does not lend itself directly to a proof-theoretic reconstruction. The theory, being declarative in a strong sense, has obscure operational aspects. Also, mainly for practical (but possibly also for theoretical) reasons, it appears to be desirable to have a version of HPSG building binary branching syntax trees. So, as a first step we present a binary version of HPSG.

3 Binary-Branching HPSG

We start off by presenting a binary version of HPSG which removes some of the unattractive features of classical HPSG. Most significantly, this version makes no use of vacuous application of rules to signs, and thus allows for signs to monotonously evolve from lexical to non-lexical status. The theory remains very close to classical HPSG in all other aspects. The binarity is mainly motivated from practical reasons; it facilitates the linking of HPSG to a logical type calculus. Binarity is by no means a strong commitment, however. Focus is on the desire to analyse a fragment of Dutch declarative main clauses, although some examples illustrate the applicability of the binary apparatus on fragments of English as well.

First, we define lexicality in terms of daughters, using common predicate notation.

- `lexical`(Sign) if `dtrs`(Sign)= $\langle\rangle$, parafrased as: Sign is lexical if Sign has zero daughters.

We then define:

- $\texttt{args}_n$(Sign) if `length`(`subcat`(Sign))= $n, n \geq 1$, parafrased as: Sign wants n arguments if the subcat list of Sign has length n (an empty list has length zero).

We refer to a functor $\mathcal{F}$ with $\texttt{args}_n$ as $\mathcal{F}_n$.

The crucial observation for languages like Dutch and English is that the amount of saturation together with the lexicality of a functor (a sign with non-empty subcat list) determines the position of the functor with respect to its argument. A post-modifier like *with pictures*, modifying a noun like *book*, follows the noun: it is non-lexical, and has $\texttt{args}_1$. Similarly, intransitive verbs — assuming they are lexicalised as VP's, i.e. non-lexical, verbal $\texttt{args}_1$ functors — follow their subjects. Semi-saturated verbal functors like *gives John* precede their objects: they are $\texttt{args}_2$ functors. We can capture the order determiner-noun by assuming that determiners subcategorise for non-maximal noun projections (like *book*, *little book with black cover*), so they are $\texttt{args}_1$; they are lexical, and precede their argument. This contrasts with the view of Pollard — Sag (1987), which analyses nouns as subcategorising for determiners.[1] So, the generalisation seems to be that:

1. Ordering effects triggered by the lexicality of functors come into play only for $\mathcal{F}_1$ functors: a lexical $\mathcal{F}_1$ is ordered before its argument; a non-lexical $\mathcal{F}_1$ is ordered after its argument.
2. A functor $\mathcal{F}_n$ where $n > 1$ is ordered before its argument.

The following LP principles make this precise:[2]

(BLP1)	$\left[\begin{array}{ll}\texttt{lex} & +\\ \texttt{args}_1 & \end{array}\right] < \alpha$
(BLP2)	$\alpha < \left[\begin{array}{ll}\texttt{lex} & -\\ \texttt{args}_1 & \end{array}\right]$
(BLP3)	$[\ \texttt{args}_n\ ] < \alpha$

To see how these principles work, consider the derivation in figure 2.

[1] We shall neglect the question on how to encode (non)-maximality of phrases here; a bar-level along the lines Cooper (1990) suggests may be necessary here.

[2] The signs in these principles are only partly specified.

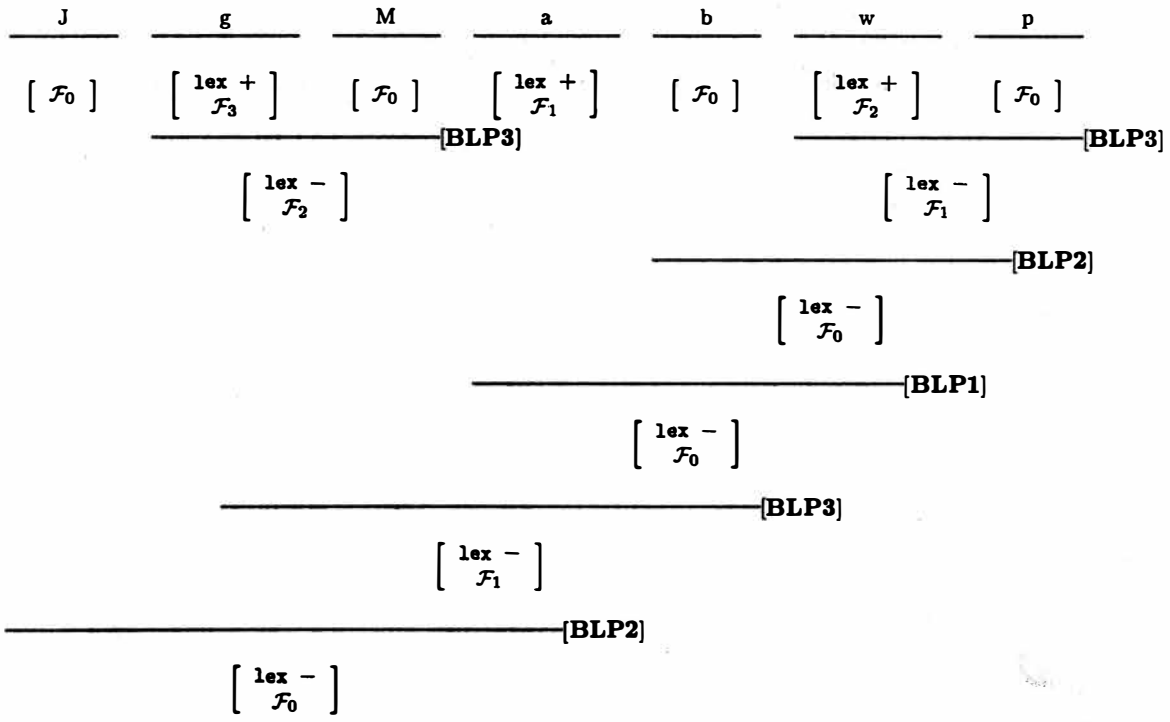


Figure 2: Sample derivation for 'John gives Mary a book with pictures'.

We also need the regular LP principle for inverted phrases:

(BLP4)	[inv +] < α

This all works fine for concatenative phenomena, i.e. the combination of two phrases under adjacency. Certain adjuncts appear to be non-concatenative, however. In Dutch, one observes:

- Jan geeft met plezier Marie een boek.
  John gives eagerly Mary a book.
- Jan geeft Marie met plezier een boek.
- Jan geeft Marie een boek met plezier.

This suggests that the phonological operation associated with certain adverbial modifiers should not be concatenation but *infixation*. Reape (1990) has made similar remarks concerning semi-free word order phenomena. We then arrive at the following LP principle

(BLP5)	[ADVMOD] ⇓ α

where $A \Downarrow B$ says that (the phonology of) A is infixed into (the phonology of) B. The non-concatenative connective $\downarrow$ was introduced in categorial grammar by Moortgat (Moortgat, 1988) for similar purposes; an expression of type $A \downarrow B$ infixes into expressions of type B to form an expression of type A (see section 4.4.2 below). ADVMOD describes the sign for a VP-level adverbial modifier, which is a sign subcategorising for a VP to yield a VP: it inherits the NP argument (the subject) its argument VP is still incomplete for. There is a little snag here: mere infixation of the adverbial phonology into the VP phonology would result in ill-formed strings where the adverb penetrates into one of the verbal arguments. For instance,

*Jan geeft de graag man een boek
John gives the with-pleasure man a book

This problem cannot be fixed by letting

phonology-values be nested lists (lists of lists) rather than *flat* lists, for instance

[Jan,[[geeft,[de,man]],[een,boek]]]

The infixation of *graag* into the VP phonology [[geeft,[de,man]],[een,boek]] will be possible only for

- graag geeft de man een boek
- geeft de man graag een boek
- geeft de man een boek graag

Deriving the well-formed

- Jan geeft graag de man een boek

now becomes hard: a rebracketing of

[[geeft,[de,man]],[een,boek]]

to

[[geeft],[[de,man],[een,boek]]]

will be necessary. So, it is not entirely clear whether the phonological operation of adverbial modifiers is not beyond simple infixation. For the moment, we leave the topic.

Complement order needs no longer be stipulated as a separate LP principle: functors now combine with one argument at a time, and the order of arguments is expressed by the order on the Subcat list.

The ID rules of original HPSG must be adapted as well; while Rules 1 and 3 can be kept, Rule 2 must now be altered to cater for generalised incompleteness: a sign having more than one item on its subcategorisation list is a well-formed sign as well.

4 Deduction for HPSG

With the binary version of HPSG we are set to give HPSG a deductive basis. First, we show that it is possible to reinterpret signs as types. Then we introduce a deductive apparatus performing type-deduction with these derived types. This calculus builds binary proof trees (*proof terms*), which are orthogonal to (binary-) HPSG derivation trees.

4.1 Signs as Formulae

We propose to view signs as *types*, or, with the Curry-Howard isomorphism in mind, as *formulae* of a certain logic. Ideas in this spirit can already be found in work of Blackburn (interpreting signs as modal formulae), Morrill and others. The concept of types has many interpretations, but one particularly apt for linguistics is that a type is a set of expressions, or, in more traditional terms, a category. Together with a set of combinatorial principles, types form an algebra of expressions over a certain domain: a type system. Essentially, these combinatorial principles constitute a *derivability relation* between sequences of types '$\rightarrow$': $A \rightarrow B$ saying that from the type sequence A the type sequence B can be derived. An example of a type system would be any syntactic algebra consisting of a set of type formation rules (e.g. the production rules in a rewrite system) and a set of syntactic categories (types) containing expressions over some alphabet of strings. More fine-grained type systems make a distinction between atomic and complex types: atomic types being monadic objects and complex types being made up from (atomic or complex) subtypes with the use of so-called type-forming connectives which serve to express combinatorial properties. Type-forming connectives are relations over the set of type symbols; a familiar example are the slashes from categorial grammar $/, \backslash$: a *functor* type X/Y combines with a type Y to its *right* to form a type X; a functor type $Y\backslash X$ combines with a type Y to its *left* to form an X. There is a nice interpretation of linguistic types as propositional formulae in a logic: atomic types T correspond to formulae T; complex types like $A\backslash B$ correspond to $A \Rightarrow_l B$, with $\Rightarrow_l$ a left-oriented version of the implication arrow $\Rightarrow$ of propositional logic. The combination of a type A with a type $A \Rightarrow B$ to a type B then becomes an instance of Modus Ponens, of which we now have two versions: $A, A \Rightarrow_l B \rightarrow B$ and $A \Rightarrow_r B, A \rightarrow B$. This, in fact, is an operationalisation of the slogan *parsing as deduction*, and is basically the central theme of categorial deduction as in Lambek calculus (Moortgat, 1988).

The intuition that signs can be interpreted as types arises from the functionality expressed by the `subcat` feature: essentially, this feature

expresses that a certain sign is *functionally (in-)complete* for one or more other signs. This immediately suggests a functional type equivalent (a functor) for these signs. Saturated signs then can be interpreted to correspond to saturated functors, or atomic types, i.e. types not being made up from a type-forming connective and one or more subtypes. HPSG's Subcat principle, which allows for the combination of a non-saturated sign with a subset of the signs it subcategorises for should then correspond to a combinatorial rule of type formation, i.e. an inference rule in a type calculus.

When we want to make a correspondence between the signs of HPSG and types of a certain kind, we immediately notice that HPSG signs encode much more information than the monadic categories of simple type systems like production grammars. A category like S, for instance, is represented in HPSG as a fine-grained specification of a verbal projection having various properties among which is an empty subcategorisation frame. Clearly, we need a more sophisticated type language than can be offered by monadic categories alone. Suppose then we switch from monadic types to types with internal structure: *predicational* types in stead of propositional types. The value of the category-determining `maj(or)` attribute should become the top-level predicate constant. As in predicate logic, types (propositions) are made up from such a predicate constant and terms as arguments of the predicate. Signs having variable values, i.e. being *underspecified* for certain attributes, correspond to (universally) quantified formulae. E.g., a partial sign like

$$\begin{bmatrix} \texttt{major} & \texttt{n} \\ \texttt{gender} & \texttt{neuter} \\ \texttt{person} & \_x \end{bmatrix}$$

with $\_x$ a variable should correspond to the type

$$\forall(\_x).[n(gender(neuter), person(\_x))].$$

The choice between universal and existential quantification is mainly motivated from considerations regarding the proof terms for quantified formulae, which will be discussed in the next section. A related motivaton is the fact that universally quantified types have a straightforward connection with Prolog literals, facilitating implementation. It is important to notice that there is no deep, 'predicate-like' meaning behind such a formula: it is just a description of a certain kind of category, in the case above having a variable spot for the person value. Sign-valued attributes, i.e. attributes taking a full sign as value, or a list of signs, are treated the same: whenever such an attribute takes a variable sign as value, universal quantification over this variable occurs. This is responsible for the second-order nature of the type language we use.

Under the logical interpretation of types as formulae, types have *proof terms* associated with them; these proof terms are the *justification* for assuming the formula is true: they correspond to proofs for the propositions the types express. These proofs are constructed in a *calculus* of inference rules, the inference rules constituting a derivability relation over type sequences (like the combinatorial rules of production systems), where this derivability relation now gets a logical interpretation as well. An alternative, quite common point of view is that proof terms are a kind of procedures (or programs) and types are the specification of what these programs do. For instance, the formula

$$\forall(\_x).[n(gender(masc), number(\_x))]$$

would be a specification of the program recognising singular and plural masculine noun phrases (this basically is what parsing is about).

A concept like *reentrancy* can easily be encoded by means of variable sharing, for example

$$\forall(\_x).[P_1(P_i(\_x), \ldots, P_n(\_x))]$$

where each P_j is a predicate symbol.

We now turn to the translation from signs to types, where we let $\dagger(S)$ yield the formula (type) equivalent of the sign S. A few words on notation: $\vec{Q}$ denotes a sequence $\forall(\_x_i)\ldots\forall(\_x_n)$ of quantifiers. The empty quantifier sequence is written as $\mathcal{Q}_0$; $\mathcal{Q}_0.\mathcal{F} = \mathcal{F}$. Further, $\vec{Q}\mathcal{Q}_0 = \mathcal{Q}_0\vec{Q} = \vec{Q}$. We use the notation

$$\Sigma \begin{bmatrix} a_i & b_i \\ & \vdots \\ a_n & b_n \end{bmatrix}$$

to refer to some sign Σ with the sign

$$\begin{bmatrix} a_i & b_i \\ & \vdots \\ a_n & b_n \end{bmatrix}$$

as a subsign. Likewise,

$$\Sigma - \begin{bmatrix} a_i & b_i \\ & \vdots \\ a_n & b_n \end{bmatrix}$$

refers to some sign Σ with the sign

$$\begin{bmatrix} a_i & b_i \\ & \vdots \\ a_n & b_n \end{bmatrix}$$

deleted from it. Furthermore, $\mathtt{var}(X)$, $\mathtt{atom}(X)$, $\mathtt{number}(X)$ express respectively that X is a variable, an atom or a number.

- $\dagger(X) := \mathcal{Q}_0.X$
  if $\mathtt{var}(X)$ **or** $\mathtt{atom}(X)$ **or** $\mathtt{number}(X)$ **or** $X = \langle\rangle$
- $\dagger(\Sigma \begin{bmatrix} \mathtt{subcat}\ \langle\rangle \\ \mathtt{maj}\ M \end{bmatrix}) := \vec{\mathcal{Q}}.M(F)$
  if $\dagger(\Sigma - \begin{bmatrix} \mathtt{subcat}\ \langle\rangle \\ \mathtt{maj}\ M \end{bmatrix}) := \vec{\mathcal{Q}}.F$
- $\dagger(\Sigma \begin{bmatrix} \mathtt{subcat}\ \langle X_1, ..., X_n\rangle \\ \mathtt{maj}\ M \end{bmatrix}) :=$
  $\vec{\mathcal{Q}}_1\vec{\mathcal{Q}}_2.A_1 \Rightarrow \ldots A_n \Rightarrow M(F)$
  if $\dagger(\Sigma - \begin{bmatrix} \mathtt{subcat}\ \langle X_1, ..., X_n\rangle \\ \mathtt{maj}\ M \end{bmatrix}) := \vec{\mathcal{Q}}_1.F$
  and $\dagger(\langle X_1, ..., X_n\rangle) := \vec{\mathcal{Q}}_2.\langle A_1, \ldots, A_n\rangle$
- $\dagger([\ A\ V\ ]) := \forall(V).A(V)$ **if** $\mathtt{var}(V)$
- $\dagger([\ A\ V\ ]) := \vec{\mathcal{Q}}.A(X_1, \ldots, X_n)$
  if $\dagger(V) := \vec{\mathcal{Q}}.\langle X_1, \ldots, X_n\rangle$
- $\dagger(\langle X_1, \ldots, X_n\rangle) := \vec{\mathcal{Q}}_1\vec{\mathcal{Q}}_2.\langle F_1, F_2, \ldots, F_n\rangle$
  if $\dagger(X_1) := \vec{\mathcal{Q}}_1.F1$ **and**
  $\dagger(\langle X_2, \ldots, X_n\rangle) := \vec{\mathcal{Q}}_2.\langle F_2, \ldots, F_n\rangle$

The crucial thing to note is that the `subcat` information of a sign is reformulated as the functional demands of a functor type: a `subcat` list of length n yields a functor with functional degree n, where n now indicates the number of arguments the functor is incomplete for.

The following example illustrates the mapping from signs to formulae. Variables are prefixed with a *don't care* '_'.

```
[syn,[[loc,[[head,[[maj,n],
                  [case,_c],
                  [nform,_n],
                  [aux,nil],
                  [inv,nil],
                  [prd,nil]]],
           [subcat,[]],
           [lex,1]]],
     [bind,_b]]]
```

then becomes

$$\forall(\_c)\forall(\_n)\forall(\_b).$$
$$[n(syn(loc(head(case(\_c), nform(\_n), aux(nil),$$
$$inv(nil), prd(nil)), lex(1)), bind(\_b)))]$$

4.2 Type deduction

Now that we have types, the question arises: what do we do with these types? In this section we show how we can interpret the HPSG apparatus of ID rules and various principles as an inference mechanism for type deduction. Before we do so, a few words on type deduction are necessary.

As mentioned in section 4.1, types have a truth-conditional interpretation: they correspond to *propositions* (formulae). This logical point of view makes it possible to identify type derivability relations with *logical* derivability relations from proof theory. A statement $A \rightarrow B$ expressing the derivability of type sequence B from type sequence A is then called a *sequent* (Gallier, 1986). A sequent $A_1, ..., A_n \rightarrow B$ can be interpreted as: the validity of the formulae $A_1, ..., A_n$ implies the validity of B; i.e., there is no model for the formulae $A_1, ..., A_n$ that is not also a model for B. The sequence $A_1, ..., A_n$ is called the *antecedent* of the sequent; the sequence B (in the present case of length 1) is called the *succedent* of the sequent. The configuration $\frac{P_1 \quad \ldots \quad P_n}{C}$ is read as: the *conclusion sequent* C is valid iff the *premise sequents* $P_1, \ldots, P_n$ are valid. As an example, here is a fragment of so-called *linear non-commutative propositional logic*. 'Linear' (Girard, 1987) means here that this logic forces 'honest' bookkeeping: we are not allowed to duplicate nor delete types during derivation. From a linguistic point of view, linearity can be used to express the fact that

the meaning of an utterance depends on the linear order of its words. Every Γ_i is a (possibly empty) type sequence; Λ is a non-empty type sequence, and X, Y, Δ are types. The comma ',' denotes non-commutative concatenation: Γ_1, Γ_2 is the concatenation of the type sequences Γ_1 and Γ_2. This entails that antecedents are essentially lists of types.

$$\mathcal{I}\frac{}{X \to X}$$

$$\mathcal{R}\Rightarrow\frac{\Gamma, X \to Y}{\Gamma \to X \Rightarrow Y}$$

$$\mathcal{R}\Rightarrow\frac{X, \Gamma \to Y}{\Gamma \to X \Rightarrow Y}$$

$$\mathcal{L}\Rightarrow\frac{\Lambda \to X \quad \Gamma_1, Y, \Gamma_2 \to \Delta}{\Gamma_1, X \Rightarrow Y, \Lambda, \Gamma_2 \to \Delta}$$

$$\mathcal{L}\Rightarrow\frac{\Lambda \to X \quad \Gamma_1, Y, \Gamma_2 \to \Delta}{\Gamma_1, \Lambda, X \Rightarrow Y, \Gamma_2 \to \Delta}$$

The $\mathcal{L}$ rules are referred to as the *left rules*; the $\mathcal{R}$ rules as the *right* rules of the calculus. Here is a proof of the theorem $A \to (A \Rightarrow B) \Rightarrow B$.

$$\mathcal{R}\Rightarrow\frac{\mathcal{L}\Rightarrow\dfrac{\mathcal{I}\dfrac{}{A \to A} \quad \mathcal{I}\dfrac{}{B \to B}}{A, A \Rightarrow B \to B}}{A \to (A \Rightarrow B) \Rightarrow B}$$

As we alluded to in section 4.1, it is possible to associate with deductions *proof terms* encoding the proofs performed; these terms are λ-terms made up from the terms associated with the types in the sequents. The λ-terms come in various kinds; the ones we discuss are either *application terms* $t(t')$, saying that the functional term t is applied to the term t'; or *abstraction terms* $\lambda v.t$, a functional term taking a term v to a term t. Terms are in either *normal form* or *non-normal form*; in the latter case, terms contain subterms $(\lambda v.t)(t')$, so-called *redexes*. The relation called β-reduction allows the simplification of such a redex to $t[t'/v]$, which means that in term t, every occurrence of v is replaced by t'. The λ-terms for these deductions have the so-called *single-bind* property: every λ-bound variable v such that $\lambda v.t$ occurs exactly once in t; so we do not have terms $\lambda v.w$ where v does not occur in w, nor terms like $\lambda v.(t(v)(v))$. We then end up with the following rules:

$$\mathcal{I}\frac{}{t : X \to t : X}$$

$$\mathcal{R}\Rightarrow\frac{\Gamma, v : X \to t : Y}{\Gamma \to \lambda v.t : X \Rightarrow Y}$$

$$\mathcal{R}\Rightarrow\frac{v : X, \Gamma \to t : Y}{\Gamma \to \lambda v.t : X \Rightarrow Y}$$

$$\mathcal{L}\Rightarrow\frac{\Lambda \to t' : X \quad \Gamma_1, t(t') : Y, \Gamma_2 \to \Delta}{\Gamma_1, t : X \Rightarrow Y, \Lambda, \Gamma_2 \to \Delta}$$

$$\mathcal{L}\Rightarrow\frac{\Lambda \to t' : X \quad \Gamma_1, t(t') : Y, \Gamma_2 \to \Delta}{\Gamma_1, \Lambda, t : X \Rightarrow Y, \Gamma_2 \to \Delta}$$

The term for the proof above would be $\lambda P.P(t)$ giving the term t as a proof for A. Once one adds the so-called Cut rule to the calculus:

$$\text{Cut}\frac{\Gamma_2 \to \Lambda \quad \Gamma_1, \Lambda, \Gamma_3 \to \Delta}{\Gamma_1, \Gamma_2, \Gamma_3 \to \Delta}$$

λ-terms in non-normal form occur as proof terms. The Cut rule expresses the transitivity of the derivability relation $\to$.

Cut-free sequent calculus for the linear fragment of propositional logic has the so-called *subformula property*: premise sequents contain all and only subformulae of the conclusion sequent. Premise sequents have lower degree in terms of type-forming connectives: they contain one connective less than the conclusion sequents. From a top-down theorem proving regime, this means a steady reduction of complexity during deduction: one starts with a 'complex' sequent containing a lot of connectives, breaking this sequent down into sequents of smaller degree, until one reaches the axiom sequents of type $A \to A$, thus settling the conjecture of the conclusion sequent. In calculi with Cut, the subformula property no longer holds, since Λ can be any type, possibly increasing the degree of the premise sequent $\Gamma_1, \Lambda, \Gamma_3 \to \Delta$. Fortunately, the Cut elimination theorem (Gentzen's **Hauptsatz** (Gentzen, 1934)) says that Cut is a derivable rule: every proof with Cut can be transformed into a Cut-free proof. Cut-elimination leads to normal-form proof terms.

Here are the sequent rules for second-order quantifier types (Morrill, 1990).

$$\mathcal{L}\forall\frac{\Gamma_1, t(t') : \Lambda[t'/\_x], \Gamma_2 \to \alpha : X}{\Gamma_1, t : \forall(\_x).\Lambda, \Gamma_2 \to \alpha : X}$$

$$\mathcal{L}\exists\frac{\Gamma_1, \pi_2(t) : \Lambda[\pi_1(t)/\_x], \Gamma_2 \to \alpha : X}{\Gamma_1, t : \exists(\_x).\Lambda, \Gamma_2 \to \alpha : X}$$

$$\mathcal{R}\forall\frac{\Gamma \to t : \Lambda}{\Gamma \to \lambda\_x.t : \forall(\_x).\Lambda}$$

$$\mathcal{R}\exists\frac{\Gamma \to t_2 : \Lambda[t_1/\_x]}{\Gamma \to \langle t_1, t_2\rangle : \exists(\_x).\Lambda}$$

For $\{\mathcal{R}\forall, \mathcal{L}\exists\}$, the condition is that $\_x$ is not free in $\Gamma, \Gamma_1, \Gamma_2$; $t[\alpha/\beta]$ is the substitution of α for β in t and $\pi_i(t)$ is the i-th projection of the pair-term t: $\pi_1(\langle\alpha, \beta\rangle) = \alpha; \pi_2(\langle\alpha, \beta\rangle) = \beta$. The functional proof terms for $\forall$-types reflect the intuitionistic idea that a proof for a proposition $\forall(\_x).\Lambda$ consists of a method for proving the proposition expressed by Λ. This gives a more plausible interpretation of proofs for universal quantification once this quantification ranges over infinite domains: a mere truth-value then seems impossible to arrive at. The pair terms for $\exists$-types say that a proof for such a formula consists of an individual (a *witness*) and a term in which this individual is substituted for the bound variable. As noted earlier. the intuitionistic quantifier terms have a nice interpretation in our syntactic type calculus: a type $\forall(\_x).\Pi$ then becomes a specification of a method (proof) recognising all expressions of type Π on the basis of any (instantiation of) $\_x$.

We shall be silent about proof terms from now on, as they do not play an evident role in parsing HPSG. They could be of use in proving meta-results about HPSG parsing, however.

Given the calculus presented above, let us establish the fragment needed to perform deduction for HPSG.

4.3 ID rules as axiom schemata.

HPSG rules describe admissible, i.e. well-formed signs. In a type-theoretic setting, they can be interpreted as type definitions, since here, signs become types. A simple way to implement these definitions, is to formulate them as *axiom schemata* in a type calculus. That is, every rule R defining the sign Σ becomes an axiom scheme:

$$R \quad \dagger(\Sigma) \to \dagger(\Sigma)$$

Every axiom sequent thus becomes an instance of an ID rule. This assures that, whenever an axiom schema is used during deduction, the type check is effective.

For binary HPSG, this results in the following axiom schemata, where $\vec{\_x}$ schematises over types

$$\vec{Q}\_x_i \Rightarrow \ldots \Rightarrow \_x_n, 1 \leq i \leq n.$$

$$\begin{array}{l}\forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\\ [\_cat(\_phon, \_syn, \_sem, \_dtrs)] \to\\ \forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\\ [\_cat(\_phon, \_syn, \_sem, \_dtrs)]\end{array}$$

$$\begin{array}{l}\forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\ \vec{\_x}\ .\\ [\_cat(\_phon, \_syn, \_sem, \_dtrs)] \to\\ \forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\ \vec{\_x}\ .\\ [\_cat(\_phon, \_syn, \_sem, \_dtrs)]\end{array}$$

It is not too hard to recognise equivalents of the HPSG rules **R1** and **R2** in these axiom schemata, once one remembers that functors now combine with their arguments one at a time: in classical HPSG, there were only two kinds of functional configurations: a functor having consumed all of its arguments (treated by **R1**) and a functor having consumed all of its arguments but one (**R2**). In $\mathcal{D}$-HPSG, many more configurations arise, generally speaking: $n - 1$ for any n-placed functor. So, where the first axiom schema restores **R1**, the second can be seen to be a generalisation of **R2** to cover *any* kind of functional incompleteness. The lexicality demand on the head daughter Rule 2 makes vanishes here; functors consume one argument at a time, and once they have consumed one, they are no longer lexical.[3]

There is another option here: the ID-rules could be compiled away by ensuring they are consequently applied to every sign and its phrasal subsigns when the lexicon is created. Although this idea entirely hides the important concept of ID rules in the process of lexicon creation, it allows for using the regular axiom scheme

[3]The demand that Rule 2 makes on the non-invertedness of the head daughter is left unexpressed here.

$$\mathcal{I}\frac{}{t : X \rightarrow t : X}$$

4.4 Principles as inference rules and conditions

The various principles of HPSG appear to be easily reconciliable with the logical setting proposed. In HPSG, they do not form a homogenous class; some principles govern the flow of information in a feature structure, others create new information (like the LP principles). This is reflected in their proof-theoretic reconstruction.

4.4.1 Head Feature Principle

The Head Feature Principle of HPSG instantiates the head features of a fresh 'mother sign' to the head features of the head daughter. The necessity for doing so vanishes in the type-theoretic HPSG equivalent. To see this, notice that in the latter, all necessary feature transport is encoded by means of variable sharing in the *type assignments* for the lexical entries. Where HPSG uses the Subcat Principle to create new sign projections, with new `compdtrs` and `subcat` values, $\mathcal{D}$-HPSG never creates new sign projections during analysis: types only become gradually more developed in the sense that more and more variable subtypes become instantiated. Therefore, the Head Feature Principle becomes totally redundant: the head features of a functor (a verb, or whatever) are preserved and developed all the way. This makes $\mathcal{D}$-HPSG in a way more lexical than original HPSG. The distinction among head daughters and their superordinating signs vanishes as well; one reasonable thing to say is that the head feature principle is 'compiled away' in the lexicon, making this distinction irrelevant. So, we can suppress the `headdtr` attribute in our signs. Another option is to keep the attribute, letting it have as value a sign which has a nil value for `headdtr`.

4.4.2 Subcat Principle

The Subcat Principle is the motor behind syntactic combination in HPSG. Basically, what it does in original HPSG is to decompose the subcat list of a non-saturated sign, transferring a (non-fixed) number of entries on the list to the `compdtrs` attribute of a fresh mother sign, thus allowing the combination of the sign with suitable complements matching the `compdtrs` value. Unification takes care of making this match by recursively descending into the mother sign.

In $\mathcal{D}$-HPSG, the Subcat Principle has a binary shape: it secures the combination of a functor $A \Rightarrow B$ and a (single) argument expression A to a result type B. As $A \Rightarrow B$ is an undirected functor, combining with an A either to its left or right to form a B, we will need two versions (left and right) of the Subcat Principle. These can be interpreted as *inference rules*, i.e. the *left rules* for the propositional connective $\Rightarrow$ for undirected implication we saw earlier. The (type) variable sharing in these rules must now be understood as demands for unification on type level:

$$\mathcal{L}\Rightarrow\frac{\Lambda \rightarrow X \quad \Gamma_1, Y, \Gamma_2 \rightarrow \Delta}{\Gamma_1, X \Rightarrow Y, \Lambda, \Gamma_2 \rightarrow \Delta}$$

The Subcat principle covers *concatenative* functors only, i.e. functors which either follow or precede their arguments. For non-concatenative functors, such as the adverbial modifiers of section 3, we cannot use the concatenative connective $\Rightarrow$.

Borrowing the connective $\downarrow$ from categorial grammar (Moortgat, 1988), then, $A \downarrow B$ is a expression wanting to penetrate in an expression of type B to form an A. The adverbial adjuncts are typed $vp \downarrow vp$, where vp is an abbreviation of a formula $n(\ldots) \Rightarrow v(\ldots)$. It turns out to be technically impossible to establish a full logic for this connective under the perspective of antecedents as lists; only the rule $\mathcal{L}\downarrow$ can be formulated.[4] See Moortgat (1988, 1990) for discussion resp. a solution. For our purposes, this is enough, however: HPSG displays partial logics (left rules only) for functional connectives. The rule becomes:

$$\mathcal{L}\downarrow\frac{\Gamma_2, \Gamma_3 \rightarrow t' : B \quad \Gamma_1, t(t') : A, \Gamma_4 \rightarrow \Lambda}{\Gamma_1, \Gamma_2, t : A \downarrow B, \Gamma_3, \Gamma_4 \rightarrow \Delta}$$

[4] A full logic for this connective would make the structural rule of Permutation: $\frac{\Gamma' \rightarrow \Delta}{\Gamma \rightarrow \Delta}\,\mathtt{permutation}(\Gamma) = \Gamma'$ derivable. This means that antecedents now become treated as multisets (sets with repetition) rather than lists, which is not desirable for linguistic purposes.

9 References

Cooper, R. (1990): "Specifiers, Complements and Adjuncts in HPSG". Unpubl. ms.

Dörre, J., I. Raasch (1991): *The Stuttgart Type Unification Formalism- User Manual.* IBM, Stuttgart.

Duffy, D.A. (1991): *Principles of automated theorem proving.* Wiley, Chichester.

Gabbay, D. (1991): *Labelled Deductive Systems*, Oxford University Press, to appear.

Gallier, J. (1986): *Logic for Computer Science*, Foundations of Automatic Theorem Proving. Harper and Row, New York.

Gentzen, G. (1934): "Untersuchungen über das logische Schliessen". In: *Math. Z.*,39:176-210, 405-431.

Girard, J.-Y. (1987): "Linear Logic". In: *Theoretical Computer Science*,50:1-102.

König, E. (1989): "Parsing as natural deduction". In: Proc. ACL, Vancouver.

Moortgat, M. (1988): *Categorial Investigations*, logic and linguistic aspects of the Lambek calculus. Foris, Dordrecht.

Moortgat, M. (1990): "Discontinuous Type Constructors". Paper presented at the workshop Categorial Grammar and Linear Logic, 2nd European Summerschool on Language, Logic and Information.

Morrill, G. (1990): "Grammar and logical types". Unpubl. ms., Edingburgh.

Pollard, C, I.Sag (1987): *Information-based syntax and semantics*, vol.1, CSLI Lecture Notes 13, Stanford.

Pollard, C, I.Sag (1992): *Information-based syntax and semantics*, vol.2, CSLI Lecture Notes, Stanford (forthcoming)

Popowich, F., C.Vogel (1990): '"A Logic-Based Implementation of Head-Driven Phrase Structure Grammar". In: *Proc. of the Third International Workshop on Natural Language Understanding and Logic Programming.* Lidinogo,Stockholm.

Raaijmakers, S. (forthcoming): *Parsing HPSG. An evaluation of several parsing strategies.. Ms.*, ITK.

Reape, M. (1990): "Getting things in order". Paper presented at the Symposium on Discontinuous Constituency, Tilburg University, January 25-27th 1990.

Roorda, D. (1991): *Resource Logics.* Dissertation, University of Amsterdam.

Shieber, S. (1986): *An introduction to unification-based approaches to grammar*, CSLI Lecture Notes 4, Stanford.

van Benthem, J. (1991): *Language in action*, Categories, Lambdas, and Dynamic Logic. Studies in logic and the foundations of mathematics, vol. 130. North-Holland, Amsterdam.

Wallen, L.A. (1990): *Automated proof-search in non-classical logics.* MIT Press, Cambridge, Mass.

where Γ_i is a type sequence of length ≥ 0, with the exception that at least one of Γ_2, Γ_3 is non-empty. Notice that this rule generalizes over incompleteness in the following way: if Γ_2 is empty, $\downarrow$ is an instance of /; if Γ_3 is empty, $\downarrow$ is an instance of \.

4.4.3 Semantics Principle

The Semantics Principle can be 'compiled away' as well, by putting in the lexicon the semantics of a sign as a product of the semantics of its daughter signs. This makes it possible to incorporate various kinds of semantics into lexical signs, for instance, a simple application semantics:

```
[ [phon,...]
  [syn,...
     [subcat,[[...[sem,X]]]]
   ...]
  [sem,f(X)],
  [dtrs,...]]
```

4.4.4 LP principles

Once a functor combines with one of its arguments to form a mobile the LP principles apply to order the functor and argument branches by ordering the respective phon values to arrive at the phon value of the mother node. LP principles can address both aspects of argument and functor, so they must be functions of a pair of types $\mathcal{T}$ to sets of types:

$$\mathcal{T} \times \mathcal{T} \longrightarrow POW(\mathcal{T})$$

In case a concatenative functor combines with its arguments, the string ordering functions yield a singleton set of result types; for non-concatenative functors, this result set often has an arity greater than one, since there is generally more than one string position for a non-concatenative functor, and each separate string position determines a new sign.

The operationalisation of LP principles in $\mathcal{D}$-HPSG is as follows. Once a functor has applied to its arguments, both functor and argument types are fed to the LP principles, which figure out the phon value of the range subtype of the functor. This entails that LP principles in $\mathcal{D}$-HPSG should operate as *side-conditions* on inference rules:[5]

$$\frac{P_1 \quad \cdots \quad P_n}{C} \textbf{ if } \mathbf{LP}_i \vee \ldots \vee \mathbf{LP}_n$$

LP principles operate on an argument type and a functor type. Here are the type-theoretic equivalents of the LP principles of binary HPSG. The notation $(\mathbf{BLP}_n)A \otimes B = C$ says that the result of applying the LP principle n to argument A and functor B is C. As before, $\overset{\Rightarrow}{\overset{X}{n}}Y$ schematises over types

$$X_i \Rightarrow \ldots X_n \Rightarrow Y,\ 0 \leq i \leq n$$

and

$$\overset{\Rightarrow}{\overset{X}{1}}Z = X \Rightarrow Z.$$

Further, inverted(X) says that X is inverted, ADVMOD(X) that X is an adverbial modifier, and $\texttt{infix}(S2, S1) = S3$ that $S3$ is the infixation of $S2$ into $S1$. Uninteresting variables are suppressed with an underscore, and quantifiers are omitted. Anticipating on the implementation, we use (Prolog) *difference list* notation for list construction: the difference list $[a, b, c] - [c]$ is equivalent to the list $[a, b]$. This is done to optimise the expression of list construction: concatenation of two lists can now be expressed via variable sharing with one unit clause:

```
conc_dl(A-B,B-C,A-C).
```

For example,

$$\texttt{conc\_dl}([a, b|C] - C, [c] - [], [a, b, c] - []).$$

(BLP1)

$$\overset{\Rightarrow}{\overset{X}{n}}.Y(\texttt{phon(S2 - S3)}, \_, \_, \_)$$
$$\otimes$$
$$\overset{\Rightarrow}{\overset{X}{1}}.Z(\texttt{phon(S1 - S2)}, \texttt{syn(loc(\_h, lex(+)), \_b)}, \texttt{\_u}, \texttt{\_w})$$
$$=$$
$$\overset{\Rightarrow}{\overset{X}{1}}.Z(\texttt{phon(S1 - S3)}, \texttt{syn(loc(\_h, lex(+)), \_b)}, \texttt{\_u}, \texttt{\_w})$$

(BLP2)

$$\overset{\Rightarrow}{\overset{X}{n}}.Y(\texttt{phon(S1 - S2)}, \_, \_, \_)$$
$$\otimes$$
$$\overset{\Rightarrow}{\overset{X}{1}}.Z(\texttt{phon(S2 - S3)}, \texttt{syn(loc(\_h, lex(-)), \_b)}, \texttt{\_u}, \texttt{\_w})$$

[5] This relates the current enterprise to Gabbay's (Gabbay, 1991) *labelled deductive systems*, where side-conditions on inference rules occur as well.

$$= \overset{\Rightarrow}{X}1.Z(\texttt{phon(S1-S3)},\texttt{syn(loc(\_h,lex(-)),\_b)},\texttt{\_u},\texttt{\_w})$$

(BLP3)

$$\overset{\Rightarrow}{X}n.Y(\texttt{phon(S2-S3)},\_,\_,\_) \otimes 2 \overset{\Rightarrow}{\leq} X n .Z(\texttt{phon(S1-S2)},\texttt{\_s},\texttt{\_t},\texttt{\_d}) = 2 \overset{\Rightarrow}{\leq} X n .Z(\texttt{phon(S1-S3)},\texttt{\_s},\texttt{\_t},\texttt{\_d}$$

(BLP4)

$$\overset{\Rightarrow}{X}n.Y(\texttt{phon(S2-S3)},\_,\_,\_) \otimes \overset{\Rightarrow}{X}m.Z(\texttt{phon(S1-S2)},\texttt{\_s},\texttt{\_t},\texttt{\_d}) = \overset{\Rightarrow}{X}m.Z(\texttt{phon(S1-S3)},\texttt{\_s},\texttt{\_t},\texttt{\_d}) \textbf{ if } \texttt{inverted}(Z(\texttt{phon(S1-S2)},\texttt{\_s},\texttt{\_t},\texttt{\_d}))$$

(BLP5)

$$\overset{\Rightarrow}{X}n.Y(\texttt{phon(S1)},\_,\_,\_) \otimes \overset{\Rightarrow}{X}m.Z(\texttt{phon(S2)},\texttt{\_v},\texttt{\_u},\texttt{\_w}) = \overset{\Rightarrow}{X}m.Z(\texttt{phon(S3)},\texttt{\_v},\texttt{\_u},\texttt{\_w}) \textbf{ if } \texttt{ADVMOD}(\overset{\Rightarrow}{X}m.Z(\texttt{phon(S2)},\texttt{\_v},\texttt{\_u},\texttt{\_w})) \textbf{ and } \texttt{infix(S2,S1)=S3}$$

4.5 Calculus

To summarize, here is the full calculus $\mathcal{D}$-HPSG. LP principles apply as discussed earlier to each inference rule as side-condition; they are suppressed below.

$$\forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\,[\_cat(\_phon,\_syn,\_sem,\_dtrs)] \rightarrow \forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\,[\_cat(\_phon,\_syn,\_sem,\_dtrs)]$$

$$\forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\overset{\Rightarrow}{x}.\,[\_cat(\_phon,\_syn,\_sem,\_dtrs)] \rightarrow \forall(\_cat)\forall(\_phon)\forall(\_syn)\forall(\_sem)\forall(\_dtrs)\overset{\Rightarrow}{x}.\,[\_cat(\_phon,\_syn,\_sem,\_dtrs)]$$

$$\mathcal{L}\Rightarrow\frac{\Gamma_1, t(t') : Y, \Gamma_2 \rightarrow \Delta}{\Gamma_1, t : X \Rightarrow Y, t' : X, \Gamma_2 \rightarrow \Delta}$$

$$\mathcal{L}\Rightarrow\frac{\Gamma_1, t(t') : Y, \Gamma_2 \rightarrow \Delta}{\Gamma_1, t' : X, t : X \Rightarrow Y, \Gamma_2 \rightarrow \Delta}$$

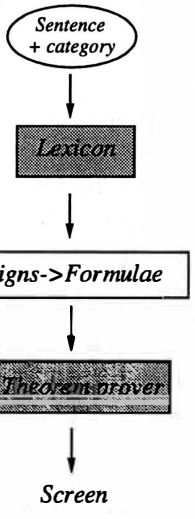
$$\mathcal{L}\forall\frac{\Gamma_1, y : \Lambda[t'/\_x], \Gamma_2 \rightarrow \alpha : X}{\Gamma_1, t : \forall(\_x).\Lambda, \Gamma_2 \rightarrow \alpha[t(t')/y] : X}$$

$$\mathcal{R}\forall\frac{\Gamma \rightarrow t : \Lambda}{\Gamma \rightarrow \lambda\_x.t : \forall(\_x).\Lambda}$$

$$\mathcal{L}\downarrow\frac{\Gamma_2, \Gamma_3 \rightarrow t' : B \quad \Gamma_1, t(t') : A, \Gamma_4 \rightarrow \Lambda}{\Gamma_1, \Gamma_2, t : A \downarrow B, \Gamma_3, \Gamma_4 \rightarrow \Delta}$$

5 Implementation

5.1 Design



The lexicon — in the overall design of the implementation pictured above — consists of lexeme-sign pairs. When a sentence _s is entered to

be analysed, first the lexemes are looked up in the lexicon. The corresponding signs are compiled to a sequence Π of formulae according to the map † of section 4.1. Then, given the result category _c, which the user is prompted to enter, the sequent $\Pi \rightarrow \_c(phon(\_s), \_syn, \_sem, \_dtrs)$ is formed, with *_syn*, *_sem*, *_dtrs* variables for syntactic, semantic and daughter information. The sequent is handed to the theorem prover, which is a meta-interpreter performing sequent deduction. Output is in both Prolog and picture format: the derived type formula for an utterance is printed on screen as a Prolog term; in a separate window the sign equivalent of the formula is drawn in standard HPSG format.

5.2 Term unification and uniform signs

Unification is a computationally expensive tool, boiling down to extensive graph inspection and merging. This cost can be saved by adopting one uniform structure for the items to be unified, i.c. signs: if we treat these signs just like rigid term structures, we could directly make use of Prolog's built-in *term unification* mechanism for unifying them. This idea entails that every lexeme in the lexicon has a fixed sign type, with standard slots and values. The structure of this sign is as follows:

```
[[phon,_p]
 [syn,[[loc,[[head,[[maj,_m],
                   [case,_c],
                   [nform,_n],
                   [vform,_v],
                   [aux,_a],
                   [inv,_i],
                   [prd,_j]]],
            [subcat,_s],
            [lex,_l]]],
      [bind,_b]]],
 [sem,_t],
 [dtrs,[[headdtr,_hd],
        [compdtrs,_cd]]]]]
```

Lexemes can be unspecified for certain attributes; these attributes then carry the atomic value **nil**.

In Prolog, variables occurring in literals are implicitly universally quantified, which means that quantifiers are removed from them. This makes it possible to use Prolog's term unification mechanism directly to instantiate values to the variables; see Duffy (1991) for discussion. So, we can simply strip quantifiers from a formula: $\vec{Q}.\mathcal{F} \mapsto \mathcal{F}$.

It is a well-known fact from the proof-theory of predicate logic that once both kinds of quantification (universal and existential) are used, deductions are not invariant under permutation of rule application: the application of quantifier elimination rules becomes order-sensitive (Wallen, 1990). This effect does not take place here, since we only have universal quantification.

5.3 Calculus in Prolog format

In Prolog format, the axioms and rules of the calculus have the following shape:

```
axiom(Name,Antecedent--->Succedent
rule(Name,Antecedent--->Succedent (if C)
```

Given a functor type $A_1 \Rightarrow \ldots A_n \Rightarrow B$, which is right-associative, i.e. $A_1 \Rightarrow (\ldots (A_n \Rightarrow B) \ldots)$, it is necessary to inspect the properties of the ultimate range subtype B when applying the LP principles. This would involve descending into the functor type, until one reaches the ultimate subtype B.

From a practical point of view, we would like to avoid such descent; we therefore notate such a functor as

$$\langle A_1, \ldots, A_n \rangle \Rightarrow B$$

where $\langle A_1, \ldots, A_n \rangle$ is a list of types. For the infix types $A \downarrow B$ we do likewise; they are usually of the form $(A_2 \Rightarrow \ldots A_n \Rightarrow B) \downarrow A_1$. To distinguish the first argument from the rest of the arguments, we write

$$\langle \#A_1, A_2, \ldots, A_n \rangle \Rightarrow B$$

In Prolog, we write `B@(_x,...,_y)` for B(*_x*, ..., *_y*). Sequents are written

$$(L_1, \ldots, L_n) \rightarrow X,$$

where each L_i is a subsequence (list) of types; type sequences of length 1 are denoted as $[T]$, with T a type or type variable. Some examples of the axiom and rule format are then:

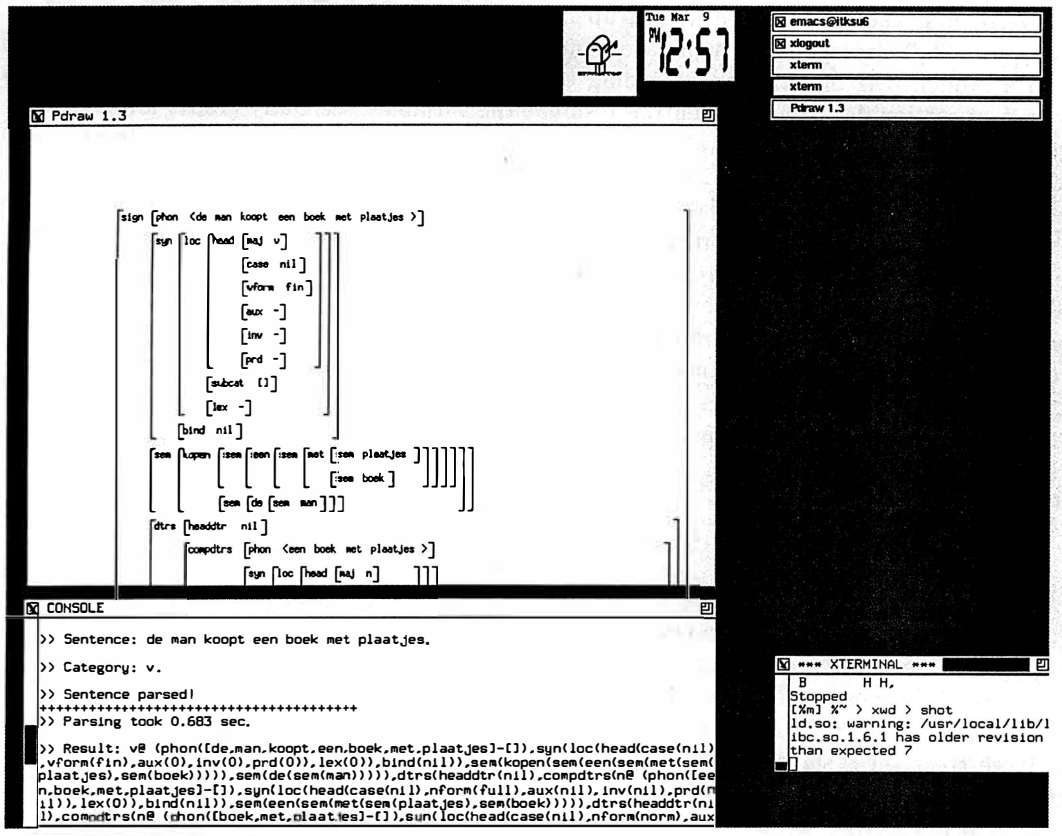


Figure 3: Screendump.

```
axiom(rule1,[C@(P,S,B,D)]--->
            [C@(P,S,B,D)]).
rule(subcat_left,
(T1,T2,[[X|T]=>B@(S1,
                  syn(loc(W1,lex(L)),
                  W2),
                S,D)],V) ---> Z
         if
T2 ---> [X]
         and
{lp(LP_Name,
 X,
 [X|T]=>B@(S1,syn(loc(W1,lex(L)),W2),
             S,D),
 [X|T]=>B@(S3,syn(loc(W1,lex(L)),W2),
             S,D))}
       and
(T1,[T=>B@(S3,syn(loc(W1,lex(0)),W2),
                S,D)],V)
  ---> Z
).
```

Notice how the lexicality value is changed to 0 here, once the functor has combined with its argument. Above we argued that the lexicality feature is a derived feature, arising from the absence or presence of daughters in a tree. Since lexical signs already have (variable) daughters in $\mathcal{D}$-HPSG, checking for lexicality could (and should) be implemented here by inspection of the daughter specifications on the functor type: if the first daughter entry in the `compdtrs` attribute list has, say, a *variable* phon value, the sign as a whole can be concluded to be lexical. For reasons of efficiency, we implement this view on lexicality by switching to non-lexicality the moment a functor combines with an argument. The variable L expresses the irrelevance of the (non-)lexicality of the functor symbol: no matter what value the functor has for `lex`, the range type will have the value - for the attribute `lex`.

5.4 Theorem prover

Various theorem proving techniques can be implemented quite easily in Prolog. As theorem prover, we use a simple sequent-proving device, implemented as follows: the prover is a set of clauses for a predicate `prove(+Goal,-Rules)`, where `Goal` is either a sequence of sequents of length minimally 1, or a structure $\{G_1,\ldots,G_n\}$ with each G_i $(1 \leq i \leq n)$ a non-sequent goal; `Rules` is a list encoding the inference rules and axioms used for proving `Goal`. Initially, `Goal` is the sequent to be proved. The predicate `prove/2` calls a routine matching the sequent against the database of inference rules, i.e. if `Goal` is of the form $X_1,\ldots,X_n \rightarrow Y$, it tries to match (resolve) the sequent against the rules and axioms of the calculus, which take the shape of $A \rightarrow B$ (**if** C). Once the match of $X_1,\ldots,X_n$ against A and Y against B has been made, the eventual premises C are attempted to prove.

```
prove({Goals},[]):- call(Goals).
prove(A--->B,[Rule|Rules]):-
        rule(Rule,(X ---> B if Y)),
        resolve(A,X),
        prove(Y,Rules).
prove(A and B,Rules):-
        prove(A,R1),
        prove(B,R2),
        conc(R1,R2,Rules).
prove(A--->B,[Ax]):-
        axiom(Ax,A1--->B),
        resolve(A,A1).
```

The linear precedence principles are (as illustrated in section 5.3) encoded as goals $\{lp_i,\ldots,lp_n\}$, to be called before entering the eventual premise sequents.

5.5 Principles

Here are some linear precedence principles. They are written as

```
lp(Name,Arg,Funct,NewFunct),
```

with `Name` the name of the principle, `Arg`, `Funct`, `NewFunct` types such that `NewFunct` has as its phonology value the ordered phonology values of `Arg` and `Funct`. Uninteresting variables are written as underscores.

```
lp(lp1,
 X@(phon(S2-S3),_,_,_),
 [X@(_,_,_,_)]=>B@(phon(S1-S2),
                   syn(loc(W1,lex(1)),W2),
                   S,D),
 [X@(_,_,_,_)]=>B@(phon(S1-S3),
                syn(loc(W1,lex(1)),W2),
                S,D)).

lp(lp4,
  X@(phon(S2-S3),_,_,_),
  [X@(_,_,_,_)|T]=>B@(phon(S1-S2),P,Q,D),
  [X@(_,_,_,_)|T]=>B@(phon(S1-S3),P,Q,D))
        :-
  inverted(B@(_,P,_,_)).
```

6 Sample lexical entries

Lexical entries are of the form

```
WORD := SIGN (← VAR_CONDITIONS)
```

with `WORD`,`SIGN` resp. a lexeme and its sign representation, and the optional `<- VAR_CONDITIONS` encoding instantiations of variables mentioned in `SIGN` (this is just done to avoid having to type very complex signs).

Some (partially specified) sample lexical entries are:

```
loopt :=
[[phon,[loopt|I]-I],
 [syn,[[loc,[[head,[[maj,v],
                   [case,nil],
                   [nform,nil],
                   [vform,fin],
                   [aux,0],
                   [inv,0],
                   [prd,0]]],
            [subcat,[X]],
              [lex,0]]],
         [bind,nil]]],
  [sem,_],
  [dtrs,[[headdtr,nil],
          [compdtrs,[X]]]]])
       <-

X= [[phon,_],
    [syn,[[loc,[[head,[[maj,n],
                      [case,nil],
                      [nform,_],
                      [vform,nil],
```

```
                        [aux,nil],
                        [inv,nil],
                        [prd,nil]]],
                  [subcat,[]],
                  [lex,0]]],
            [bind,_]]],
    [sem,_],
    [dtrs,_]].

de :=
[[phon,[de|I]-I],
 [syn,[[loc,[[head,[[maj,n],
                    [case,nil],
                    [nform,_],
                    [vform,nil],
                    [aux,nil],
                    [inv,nil],
                    [prd,nil]]],
                [subcat,[X]],
                 [lex,1]]],
       [bind,nil]]],
 [sem,_],
 [dtrs,[[headdtr,nil],
        [compdtrs,[X]]]]])
    <-

X= [[phon,_],
    [syn,[[loc,[[head,[[maj,n],
                       [case,nil],
                       [nform,_],
                       [vform,nil],
                       [aux,nil],
                       [inv,nil],
                       [prd,nil]]],
                  [subcat,[]],
                  [lex,1]]],
            [bind,_]]],
    [sem,_],
    [dtrs,_]].

man :=
[[phon,[man|I]-I],
 [syn,[[loc,[[head,[[maj,n],
                    [case,nil],
                    [nform,norm],
                    [vform,nil],
                    [aux,nil],
                    [inv,nil],
                    [prd,nil]]],
          [subcat,[]],
          [lex,1]]],
       [bind,_]]],
 [sem,_],
 [dtrs,[[headdtr,nil],
        [compdtrs,[]]]]]).
```

7 Performance

The following overview lists real-time parsing results for a small set of Dutch sentences. The results were generated by a compiled Prolog executable version of the program, running under X-windows on a SUN SPARCstation 1.

The sentences and their word-by-word translations are:

1. jan loopt.
   john walks.
2. de man loopt.
   the man walks.
3. de man loopt graag.
   the man walks gladly.
4. jan heeft hard gelopen.
   john has fast walked.
5. jan slaat de man.
   john hits the man.
6. john slaat graag de hond.
   john hits gladly the dog.
7. de man koopt een boek met plaatjes.
   the man buys a book with pictures.
8. jan geeft marie de hond.
   john gives mary the dog.
9. jan geeft marie een boek met plaatjes.
   john gives mary a book with pictures.
10. jan geeft marie graag een boek.
    john gives mary gladly a book.
11. dat jan de hond slaat.
    that john the dog hits.

```
>> Sentence: jan loopt.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.000 sec.

>> Sentence: de man loopt.
```

```
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.000 sec.

>> Sentence: de man loopt graag.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.050 sec.

>> Sentence: jan heeft hard gelopen.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.050 sec.

>> Sentence: jan slaat de man.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.067 sec.

>> Sentence: jan slaat graag de hond.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.450 sec.

>> Sentence: de man koopt een boek met
             plaatjes.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.683 sec.

>> Sentence: jan geeft marie de hond.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.117 sec.

>> Sentence: jan geeft marie een boek
             met plaatjes.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 2.500 sec.

>> Sentence: jan geeft marie graag een
             boek.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 1.316 sec.

>> Sentence: dat jan de hond slaat.
>> Category: v.
>> Sentence parsed!
++++++++++++++++++++++++++++++++++++++
>> Parsing took 0.017 sec.
```

As can be concluded from the output presented in the previous section, performance is relatively good. Sentences taking a lot of time (say, over 1 second), invariably contain at least one adverbial modifier, or involve an NP closure problem. For instance, in

- John gives Mary a book with pictures

the phrase 'John gives Mary a book' can be erroneously analysed as a sentence before the PP 'with pictures' is attached to 'a book'. Once the parser detects the remaining phrase 'with pictures', it will have to backtrack and redo a lot of work. The bad performance is a consequence of the sequent formalism: for any configuration

$$X_1 \Rightarrow X_2, X_1, X_1 \Rightarrow X_1$$

where each X_i is distinct, the analysis

$$((X_1 \Rightarrow X_2, X_1), X_1 \Rightarrow X_1)$$

is tried. One idea would be to employ a well-formed substring table encoding intermediate parsing results, to avoid having to reparse too much once the parser starts backtracking.

Generally speaking, weak performance for long sentences is not surprising, since the various inference rules allow for blind alleys in the left premise deduction, by instantiating wrong subsequences of the antecedent to the factor reducing to an argument type. This is a direct consequence of the non-deterministic nature of the procedure decomposing the antecedent into contexts around a functional type. The problem can be fixed by introducing so-called *proof invariants* (van Benthem, 1991) into the theorem prover. Proof invariants are structural validities for antecedent-succedent pairs, which serve to prune irrelevant options from the search space. The attractive feature of the current setting is that *any* optimalisation coming from proof theory can be used to optimise the parser.

8 Concluding remarks

We have shown that it is possible to give HPSG a deductive basis. The binary version of HPSG we have proposed, has been demonstrated to correspond to a fragment of second-order linear logic. The binarity of this HPSG dialect, which is faithful to classical HPSG in all other respects, is motivated from practical rather than theoretical reasons; in fact, the current approach is open to any version of HPSG. The parser we developed is, although relatively fast, in need of further optimalisation; the use of proof invariants may help to reduce the search space. Also, recently developed low-complexity theorem proving techniques such as *proof nets* (Roorda, 1991), may be of use here. Returning to the five desiderata of section 1, then, the last item, "The parser should have reasonable time/space complexity" has not fully been met yet.

Acknowledgements

I thank my colleagues René Ahn, Miriam Mulders, Gerrit Rentier and Leon Verschuur for fruitful discussion and taking an active interest in the enterprise.

Stochastic Lexicalized Context-Free Grammar

Yves Schabes and Richard C. Waters

Mitsubishi Electric Research Laboratories
201 Broadway, Cambridge, MA 02139
email: {schabes|dick}@merl.com

Abstract

Stochastic lexicalized context-free grammar (SLCFG) is an attractive compromise between the parsing efficiency of stochastic context-free grammar (SCFG) and the lexical sensitivity of stochastic lexicalized tree-adjoining grammar (SLTAG). SLCFG is a restricted form of SLTAG that can only generate context-free languages and can be parsed in cubic time. However, SLCFG retains the lexical sensitivity of SLTAG and is therefore a much better basis for capturing distributional information about words than SCFG.

1 Motivation

The application of stochastic techniques to syntax modeling has recently regained popularity. Most of the work in this area has tended to emphasize one or the other of the following two goals. The first goal is to capture as much distributional information about words as possible. The second goal is to capture as many of the hierarchical constraints inherent in natural languages as possible. Unfortunately, these two goals have been more or less incompatible to date.

Early stochastic proposals such as Markov Models, N-gram models [2, 14] and Hidden Markov Models [7] are very effective at capturing simple distributional information about adjacent words. However, they cannot capture long range distributional information nor the hierarchical constraints inherent in natural languages.

Stochastic context-free grammar (SCFG) [1, 3, 5] extends context-free grammar (CFG) by associating each rule with a probability that controls its use. Each rule is associated with a single probability that is the same for all the sites where the rule can be applied.

SCFG captures hierarchical information just as well as CFG; however, it does not do a good job of capturing distributional information about words. There are at least two reasons for this. First, many rules do not contain any words and therefore the associated probabilities do not have any direct link to words. Second, distributional phenomena that involve the application of two or more rules do not have a direct link to any of the stochastic parameters of SCFG, because the probabilities apply only to single rules.

It has been observed in practice that SCFG performs worse than non-hierarchical approaches. This has lead many researchers to believe that simple distributional information about adjacent words is the most important single source of information. In the absence of a formalism that adequately combines this information with other kinds of information, the emphasis in research has been on simple non-hierarchical statistical models of words, such as word N-gram models.

Recently, it has been suggested that stochastic lexicalized tree-adjoining grammar (SLTAG) [8, 9] may be able to capture both distributional and hierarchical information. An SLTAG grammar consists of a set of trees each of which contains one or more lexical items. These elementary trees can be viewed as the elementary clauses (including their transformational variants) in which the lexical items participate. The elementary trees are combined by substitution and adjunction. Each possible way of combining two trees is associated with a probability.

Since it is based on tree-adjoining grammar (TAG), SLTAG can capture some kinds of hier-

archical information that cannot be captured by SCFG. However, the key point of comparison between SLTAG and SCFG is that since SLTAG is lexicalized and uses separate probabilities governing each possible combination of trees, each probability is directly linked to a pair of words. This makes it possible to represent a great deal of distributional information about words.

Unfortunately, the statistical algorithms for SLTAG [9] require much more computational resources than the ones for SCFG. For instance, the algorithms for estimating the stochastic parameters and determining the probability of a string require in the worst case $O(n^6)$-time for SLTAG [9] but only $O(n^3)$-time for SCFG [3].

Stochastic lexicalized context-free grammar (SLCFG) is a restricted form of SLTAG that retains most of the advantages of SLTAG without requiring any greater computational resources than SCFG. SLTAG restricts the elementary trees that are possible and the way adjunction can be performed. These restrictions limit SLCFG to producing only context-free languages and allow SLCFG to be parsed in $O(n^3)$-time in the worst case. However, SLCFG retains most of the key features of SLTAG enumerated above. In particular, the probabilities in SLCFG are directly linked to pairs of words.

SLCFG is a stochastic extension of lexicalized context-free grammar (LCFG) [12, 13]. The following sections, introduce LCFG, define the stochastic extension to SLCFG, present an algorithm that can determine the probability of a string generated by an SLCFG in $O(n^3)$-time, and discuss the algorithms needed to train the parameters of an SLCFG.

2 LCFG

Lexicalized context-free grammar (LCFG) [12, 13] is a tree generating system that is a restricted form of lexicalized tree-adjoining grammar (LTAG) [4]. The grammar consists of two sets of trees: initial trees, which are combined by substitution and auxiliary trees, which are combined by adjunction. An LCFG is lexicalized because every initial and auxiliary tree is required to contain a terminal symbol on its frontier.

Definition 1 An *LCFG* is a five-tuple (Σ, NT, I, A, S), where Σ is a set of terminal symbols, NT is a set of non-terminal symbols, I and A are finite sets of finite trees labeled by terminal and non-terminal symbols, and S is a distinguished non-terminal start symbol. The set $I \cup A$ is referred to as the elementary trees.

The interior nodes in each elementary tree are labeled by non-terminal symbols. The nodes on the frontier of each elementary tree are labeled with terminal symbols, non-terminal symbols, and the empty string (ε). At least one frontier node is labeled with a terminal symbol. With the possible exception of one (see below), the non-terminal symbols on the frontier are marked for substitution. (By convention, substitutability is indicated in diagrams by using a down arrow ($\downarrow$).)

The difference between auxiliary trees and initial trees is that each auxiliary tree has exactly one non-terminal frontier node that is marked as the foot. The foot must have the same label as the root. (By convention, the foot of an auxiliary tree is indicated in diagrams by using an asterisk ($*$).) The path from the root of an auxiliary tree to the foot is called the *spine.*

Auxiliary trees in which every non-empty frontier node is to the left of the foot are called *left* auxiliary trees. Similarly, auxiliary trees in which every non-empty frontier node is to the right of the foot are called *right* auxiliary trees. Other auxiliary trees are called *wrapping* auxiliary trees.[1]

LCFG does not allow adjunction to apply to foot nodes or nodes marked for substitution. LCFG allows the adjunction of a left auxiliary tree and a right auxiliary tree on the same node. However, LCFG does not allow the adjunction of either two left or two right auxiliary trees on the same node.

Crucially, LCFG does not allow wrapping auxiliary trees. It does not allow elementary wrapping auxiliary trees, and it does not allow the adjunction of two auxiliary trees, if the result would be a wrapping auxiliary tree.

Figure 1, shows seven elementary trees that might appear in an LCFG for English. The trees containing 'boy', 'saw', and 'left' are initial trees. The remainder are auxiliary trees.

[1] In [13] these three kinds of auxiliary trees are referred to differently as right recursive, left recursive, and centrally recursive, respectively.

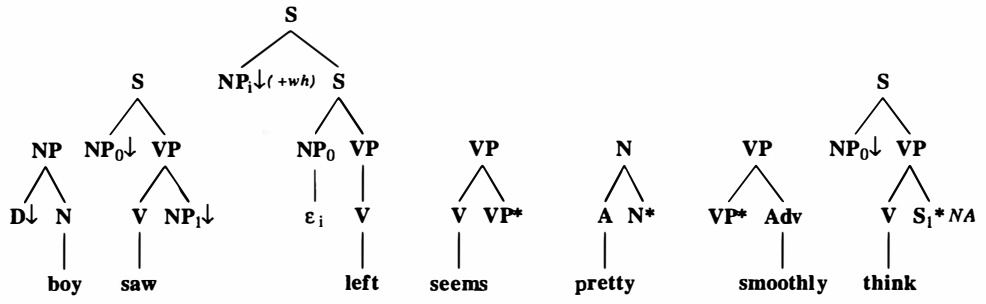


Figure 1: Example LCFG trees.

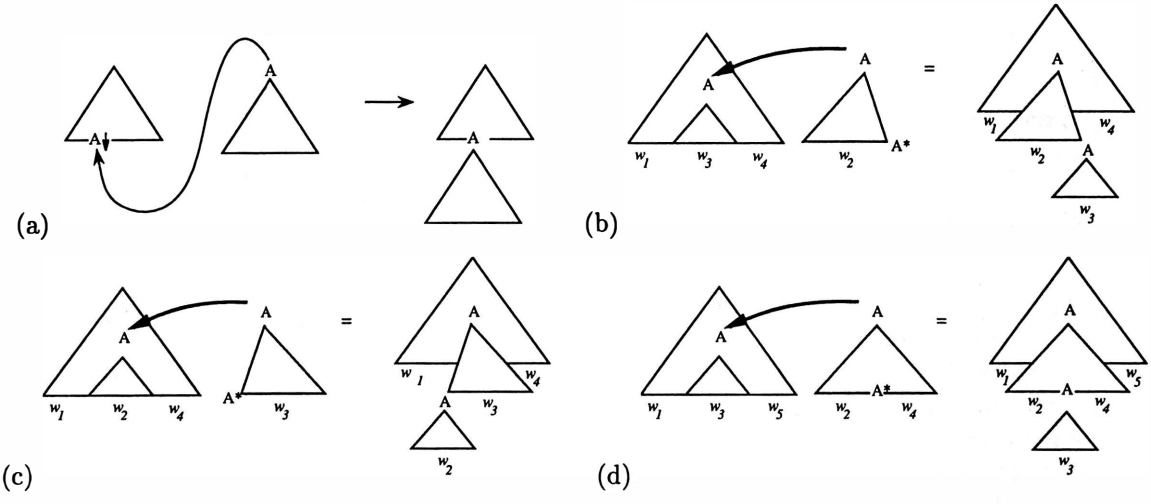


Figure 2: Tree combination: (a) substitution, (b) left adjunction, (c) right adjunction, and (d) wrapping adjunction, which is not allowed by SLCFG.

An LCFG derivation must start with an initial tree rooted in S. After that, the tree can be repeatedly extended using substitution and adjunction. A derivation is complete when every frontier node is labeled with a terminal symbol.

As illustrated in Figure 2a, substitution replaces a node marked for substitution with a copy of an initial tree.

Adjunction inserts a copy of an auxiliary tree T into another tree at an interior node η that has the same label as the root (and therefore foot) of T. In particular, η is replaced by a copy of T and the foot of the copy of T is replaced by the subtree rooted at η. The adjunction of a left auxiliary tree is referred to as left adjunction (see Figure 2b). The adjunction of a right auxiliary tree is referred to as right adjunction (see Figure 2c).

LCFG's prohibition on wrapping auxiliary trees can be rephrased solely in terms of elementary trees. To start with, there must be no elementary wrapping auxiliary trees. In addition, an elementary left (right) auxiliary tree cannot be adjoined on any node that is on the spine of an elementary right (left) auxiliary tree. Further, no adjunction whatever is permitted on a node η that is to the right (left) of the spine of an elementary left (right) auxiliary tree T. (Note that for T to be a left (right) auxiliary tree, every frontier node subsumed by η must be labeled with ε.)

Tree adjoining grammar formalisms typically forbid adjunction on foot nodes and substitution nodes. In addition, they typically forbid multiple adjunctions on a node. However, in the case of LCFG, it is convenient to relax this latter restriction slightly by allowing right and left adjunction on a node, but at most once each. (Due to the other restrictions placed on LCFG, this relaxation

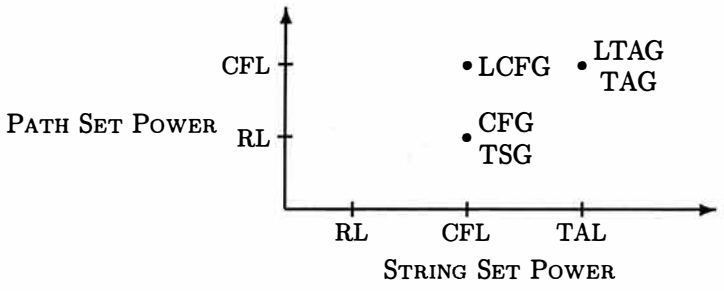


Figure 3: The tree and string complexity of LCFG and several other formalisms

increases the trees that can be generated without increasing the ambiguity of derivations.)

2.1 Comparisons

The only important difference between LCFG and LTAG is that LTAG allows both elementary and derived wrapping auxiliary trees. The importance of this is that wrapping adjunction (see Figure 2d) encodes string wrapping and is therefore context sensitive in nature. In contrast, left and right adjunction (see Figures 2b & 2c) merely support string concatenation. As a result, while LTAG is context sensitive in nature, LCFG is limited to generating only context-free languages.

To see that LCFG can only generate context-free languages, consider that any LCFG G can be converted into a CFG generating the same strings in two steps as follows. First, G is converted into a tree substitution grammar (TSG) G' that generates the same strings. Then, this TSG is converted into a CFG G''.

A TSG is the same as an LCFG (or LTAG) except that there cannot be any auxiliary trees. To create G' first make every initial tree of G be an initial tree of G'. Next, make every auxiliary tree T of G be an initial tree of G'. When doing this, relabel the foot of T with ε (turning T into an initial tree). In addition, let A be the label of the root of T. If T is a left auxiliary tree, rename the root to A_L; otherwise rename it to A_R.

To complete the creation of G' alter every node η in every initial tree in G' as follows: Let A be the label of η. If left adjunction is possible at η, add a new first child of η labeled A_L, mark it for substitution, and add a tree corresponding to $A_L \rightarrow \varepsilon$ if one does not already exist. Right adjunction is handled analogously by adding a new last child of η labeled A_R and insuring the existance of a tree corresponding to $A_R \rightarrow \varepsilon$.

The TSG G' generates the same strings as G, because all cases of adjunction have been changed into equivalent substitutions. Note that the transformation would not work if LCFG allowed wrapping auxiliary trees. The TSG G' can be converted into a CFG G'' by flattening each tree in G' into a context-free rule that expands the root of the tree into the frontier in one step.

Although the string sets generated by LCFG are the same as those generated by CFG, LCFG is capable of generating more complex sets of trees than CFG. In particular, it is interesting to look at the path sets of the trees generated. (The path set of a grammar is the set of all paths from root to frontier in the trees generated by the grammar. The path set is a set of strings over $\Sigma \cup NT \cup \{\varepsilon\}$.)

The path sets for CFG (and TSG) are regular languages [15]. In contrast, just as for LTAG and TAG, the path sets for LCFG are context-free languages. To see this, consider that adjunction makes it possible to embed a sequence of nodes (the spine of the auxiliary tree) in place of a node on a path. Therefore, from the perspective of the path set, auxiliary trees are analogous to context-free productions.

Figure 3 summarizes the relationship between LCFG and several other grammar formalisms. The horizontal axis shows the complexity of strings that can be generated by the formalisms, i.e., regular languages (RL), context-free languages (CFL), and tree adjoining languages (TAL). The vertical axis shows the complexity of the path sets that can be generated.

CFG (and TSG) create context-free languages, but the path sets they create are regular languages. LTAG and TAG generate tree adjoining languages and have path sets that are context-free languages. LCFG is intermediate in nature. It can only generate context-free languages, but has path sets that are also context-free languages.

2.2 LCFG lexicalizes CFG

As shown in [12, 13] LCFG lexicalizes CFG without changing the trees derived. Further, a constructive procedure exists for converting any CFG G into an equivalent LCFG G'.

The fact that LCFG lexicalizes CFG is significant, because every other method for lexicalizing CFGs without changing the trees derived requires context-sensitive operations [4] and therefore dramatically increases worst case processing time.

As shown in [12, 13] (and in Section 4) LCFG can be parsed in the worst case just as quickly as CFG. Since LCFG is lexicalized, it is expected that it can be parsed much faster than CFG in the typical case.

3 Stochastic LCFG

The definition of stochastic lexicalized context-free grammar (SLCFG) is the same as the definition of LCFG except that probabilities are added that control the combination of trees by adjunction and substitution.

Definition 2 An *SLCFG* is an 11-tuple (Σ,NT, I,A,S,P_I,P_S,P_L,P_{NL},P_R,P_{NR}), where (Σ,NT,I, A,S) is an LCFG and P_I, P_S, P_L, P_{NL}, P_R, and P_{NR} are statistical parameters as defined below.

For every root ρ of an initial tree, $P_I(\rho)$ is the probability that a derivation starts with the tree rooted at ρ. It is required that:

$$\sum_{\rho} P_I(\rho) = 1$$

Note that $P_I(\rho) \neq 0$ if and only if ρ is labeled S.

For every root ρ of an initial tree and every node η that is marked for substitution, $P_S(\rho,\eta)$ is the probability of substituting the tree rooted at ρ for η. For each η it is required that:

$$\sum_{\rho} P_S(\rho,\eta) = 1$$

For every node η in every elementary tree, $P_{NL}(\eta)$ is the probability that left adjunction will not occur on η. For every root ρ of a left auxiliary tree, $P_L(\rho,\eta)$ is the probability of adjoining the tree rooted at ρ on η. For each η it is required that:

$$P_{NL}(\eta) + \sum_{\rho} P_L(\rho,\eta) = 1$$

$P_{NL}(\eta) = 0$ if and only if left adjunction on η is obligatory.

The parameters $P_{NR}(\eta)$ and $P_R(\rho,\eta)$ control right adjunction in an exactly analogous way.

An SLCFG derivation is described by the initial tree it starts with, together with the sequence of substitution and adjunction operations that take place. The probability of a derivation is defined as the product of: the probability P_I of starting with the given tree, the probabilities P_S, P_L, and P_R of the operations that occurred, and the probabilities P_{NL} and P_{NR} of adjunction not occurring at the places where it did not occur.

The probability of a string is the sum of the probabilities of all the different ways of deriving it. A most likely derivation of a string is a derivation that has as large a probability as any other derivation for the string. The probability of a tree generated by an SLCFG for a string is the sum of the probabilities of every way of deriving the tree. (Unlike in SCFG, in SLCFG there can be more than one way to derive a given tree.) A most likely tree generated for a string is a tree whose probability is as large as any other tree generated for the string. (Note that a most likely derivation need not generate a most likely tree.)

4 Parsing SLCFG

Since SLCFG is a restricted case of SLTAG, the $O(n^6)$-time SLTAG parser [9] can be used for parsing SLCFG. Further, it can be straightforwardly modified to require at most $O(n^4)$-time when applied to SLCFG. However, this does not take full advantage of the context-freeness of SLCFG.

This section demonstrates that SLCFG can be parsed in $O(n^3)$-time by exhibiting a CKY-style

bottom-up algorithm for computing the probability assigned to a string by an SLCFG. This algorithm can be trivially modified to extract a most probable derivation of the given string. More efficient SLCFG processors can be based on the Earley style LCFG recognizer presented in [12].

4.1 Terminology

Suppose that G is an SLCFG and that $a_1 \cdots a_n$ is an input string. Let η be a node in an elementary tree (identified by the name of the tree and the position of the node in the tree).

Label(η) $\in \Sigma \cup NT \cup \varepsilon$ is the label of the node. The predicate IsInitialRoot(η) is true if and only if η is the root of an initial tree. Parent(η) is the node that is the parent of η or $\perp$ if η has no parent. FirstChild(η) is the node that is the leftmost child of η or $\perp$ if η has no children. Sibling(η) is the node that is the next child of the parent of η (in left to right order) or $\blacktriangle$ if there is no such node.

The predicate Substitutable(ρ, η) is true if and only if η is marked for substitution and ρ is the root of an initial tree that can be substituted for η. The predicate Radjoinable(ρ, η) is true if and only if ρ is the root of an elementary right auxiliary tree that can adjoin on η. The predicate Ladjoinable(ρ, η) is true if and only if ρ is the root of an elementary left auxiliary tree that can adjoin on η.

The concept of *covering* is critical to the bottom-up algorithm shown below. Informally speaking, a node η covers a string if and only if the string can be derived starting from η.

More precisely, for every node η in every elementary tree in G, let T' be a copy of the subtree of T that is rooted at η. Extend T' by adding a new root whose only child is the original root of T'. Label the new root of T' with a unique new symbol S'. If there is a node on the frontier of T' that is marked as the foot, relabel this node with ε. This converts T' into an initial tree. Let G_η be an SLCFG that is identical to G except that T' is introduced as an additional initial tree and the start symbol of G_η is S'. The probabilities associated with the the interior nodes of T' are identical to those for the corresponding nodes in T. The probabilities for the root of T' are $P_S = P_L = P_R = 0$, $P_{NL} = P_{NR} = 1$, and crucially $P_I = 1$. $P_I = 0$ for the other initial trees.

The node η covers a string $a_1 \cdots a_n$ with probability p in G if and only if the probability of $a_1 \cdots a_n$ in G_η is p. The node η covers a string $a_1 \cdots a_n$ without left (right) adjunction with probability p in G if and only if the probability of $a_1 \cdots a_n$ in G_η is p without considering derivations where left (right) adjunction occurs on the original root of T'.

(Note that if η is a foot node, T' is an empty tree. The only string covered by η is the empty string; however, the empty string is covered with probability 1, because the empty string is the only string derived by G_η.)

4.2 A bottom-up Algorithm

We can assume without loss of generality that every node in $I \cup A$ has at most two children. (By adding new nodes, any SLCFG can be transformed into an equivalent SLCFG satisfying this condition. This transformation can be readily reversed after parsing has been completed.)

The algorithm stores triples of the form $[\eta, code, p]$ in an $n \times n$ array C. In a triple, *code* is a set over the universe L (for left adjunction) and R (for right adjunction). The fact that $[\eta, code, p] \in C[i,k]$ means that η accounts for the substring $a_{i+1} \cdots a_k$ with probability p. More precisely, for every node η in every elementary tree in G, the algorithm guarantees that when the computation concludes:

- $[\eta, \emptyset, p] \in C[i,k]$ if and only if η covers $a_{i+1} \cdots a_k$ with probability p without left or right adjunction.
- $[\eta, \{L\}, p] \in C[i,k]$ if and only if η covers $a_{i+1} \cdots a_k$ with probability p without right adjunction.
- $[\eta, \{R\}, p] \in C[i,k]$ if and only if η covers $a_{i+1} \cdots a_k$ with probability p without left adjunction.
- $[\eta, \{L, R\}, p] \in C[i,k]$ if and only if η covers $a_{i+1} \cdots a_k$ with probability p.

The process starts by placing each foot node and each frontier node that is labeled with the empty string in every cell $C[i,i]$ with probability one. This signifies that they each cover the empty string at all positions. The initialization also puts each terminal node η in every cell $C[i, i+1]$ where

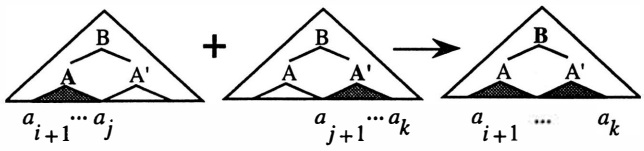


Figure 4: Sibling concatenation.

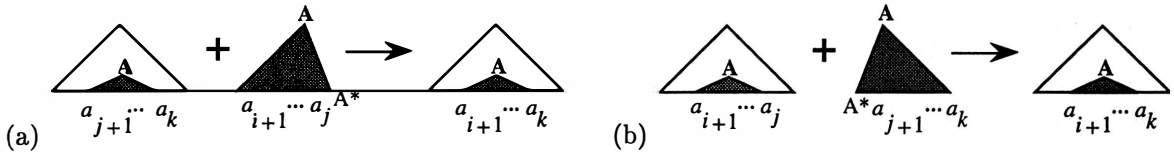


Figure 5: (a) Left concatenation and (b) right concatenation.

```
Procedure Probability(a_1 ··· a_n)
 begin
  for i = 0 to n
   for all foot nodes φ in A, Add(φ, ∅, i, i, 1)
   for all frontier nodes η in A ∪ I where Label(η) = ε, Add(η, ∅, i, i, 1)
  for i = 0 to n − 1
   for all frontier nodes η in A ∪ I where Label(η) = a_{i+1}, Add(η, ∅, i, i + 1, 1)
  for d = 0 to n
   for i = 0 to n − d
    set k = i + d
    for j = i to k
     for all nodes η in G
      if [η, {L, R}, p_1] ∈ C[i, j] and [Sibling(η), {L, R}, p_2] ∈ C[j, k]
       then Add(Parent(η), ∅, i, k, p_1 × p_2)
     for all nodes ρ and η in G where Ladjoinable(ρ, η)
      if [ρ, {L, R}, p_1] ∈ C[i, j] and [η, code, p_2] ∈ C[j, k] and L ∉ code
       then Add(η, L ∪ code, i, k, p_1 × p_2 × P_L(ρ, η))
     for all nodes ρ and η in G where Radjoinable(ρ, η)
      if [η, code, p_1] ∈ C[i, j] and R ∉ code and [ρ, {L, R}, p_2] ∈ C[j, k]
       then Add(η, R ∪ code, i, k, p_1 × p_2 × P_R(ρ, η))
  p = 0
  for all nodes ρ in G where IsInitialRoot(ρ) and Label(ρ) = S
   if [ρ, {L, R}, p_0] ∈ C[0, n] then p = p + p_0 × P_I(ρ)
  return p
 end
Procedure Add(η, code, i, k, p)
 begin
  if [η, code, p'] ∈ C[i, k] for some p' then update [η, code, p'] in C[i, k] to [η, code, p' + p]
  else C[i, k] := C[i, k] ∪ [η, code, p]
  if code = {L, R} then
   if FirstChild(Parent(η)) = η and Sibling(η) = ⊥ then Add(Parent(η), ∅, i, k, p)
   for each node φ such that Substitutable(η, φ), Add(φ, ∅, i, k, p × P_S(η, φ))
  if L ∉ code then Add(η, L ∪ code, i, k, p × P_NL(η))
  if R ∉ code then Add(η, R ∪ code, i, k, p × P_NR(η))
 end
```

Figure 6: A procedure for computing the probability of a string given an SLCFG.

η is labeled a_{i+1} with probability one. The algorithm then considers all possible ways of combining matched substrings into longer matched substrings—it fills the upper diagonal portion of the array $C[i,k]$ $(0 \leq i \leq k \leq n)$ for increasing values of $k-i$.

Two observations are central to the efficiency of this process. Since every auxiliary tree (elementary and derived) in SLCFG is either a left or right auxiliary tree, the substring matched by a tree is always a contiguous string. Further, when matched substrings are combined, the algorithm only has to consider adjacent substrings. (In SLTAG, a tree with a foot can match a pair of strings that are not contiguous—one left of the foot and one right of the foot.)

There are three situations where combination of matched substrings is possible: sibling concatenation, left concatenation, and right concatenation.

As illustrated in Figure 4, sibling concatenation combines the substrings matched by two sibling nodes into a substring matched by their parent. In particular, suppose that there is a node η_0 (labeled B in Figure 4) with two children η_1 (labeled A) and η_2 (labeled A'). If $[\eta_1, \{L,R\}, p_1] \in C[i,j]$ and $[\eta_2, \{L,R\}, p_2] \in C[j,k]$ then $[\eta_0, \emptyset, p_1 \times p_2] \in C[i,k]$.

Left concatenation (see Figure 5a) combines the substring matched by a left auxiliary tree with the substring matched by a node the auxiliary tree can adjoin on. Right concatenation (see Figure 5b) is analogous.

The algorithm (see Figure 6) is written in two parts: a main procedure Probability$(a_1 \cdots a_n)$ and a subprocedure Add$(\eta, code, i, k)$, which adds the triple $[\eta, code, p]$ into $C[i,k]$.

The main procedure repeatedly scans the array C, building up longer and longer matched substrings until it determines all the S-rooted derived trees that match the input. The purpose of the codes ($\{L,R\}$ etc.) is to insure that left and right adjunction can each be applied at most once on a node. The procedure could easily be modified to account for other constraints on the way derivation should proceed, such as those suggested for LTAGs [11].

The procedure Add enters a triple $[\eta, code, p]$ into $C[i,k]$. If some other triple $[\eta, code, p']$ is already present in $C[i,k]$, then the probability p' is updated to $p'+p$ to reflect the fact that an additional derivation of $a_{i+1} \cdots a_k$ has been found. Otherwise, a new triple $[\eta, code, p]$ is added to $C[i,k]$.

The procedure Add also propagates information from one triple to another in situations where the length of the matched string is not increased—i.e., when a node is the only child of its parent, when substitution occurs, and when adjunction is not performed.

The $O(n^3)$ complexity of the algorithm follows from the three nested induction loops on d, i and j. (Although the procedure Add is defined recursively, the number of pairs added to C is bounded by a constant that is independent of sentence length.)

The algorithm does not depend on the fact that SLCFG is lexicalized—it would work equally well if were not lexicalized. If the sum $p'+p$ on the third line of the Add procedure is changed to $\max(p', p)$ the algorithm computes the probability of a most probable derivation. By keeping a record of every attempt to enter a triple into a cell of the array C, one can extend the algorithm so that derivations and therefore the trees they generate can be rapidly recovered.

5 Training an SLCFG

In the general case, the training algorithm for SCFG [5] requires $O(n^3)$-time for each sentence of length n. A training algorithm for SLCFG can be constructed that achieves these same worst case bounds.

To start with, since SLCFG is a restricted case of stochastic lexicalized tree-adjoining grammar (SLTAG), the $O(n^6)$-time inside-outside reestimation algorithm for SLTAG [9] can be used for estimating the parameters of an SLCFG given a training corpus. Straightforward modifications lead to an $O(n^4)$-time algorithm for training an SLCFG. However, this alone does not achieve the full potential of SLCFG.

The same basic construction that underlies the algorithm in the last section can be used as the basis for an $O(n^3)$ inside-outside training algorithm for SLCFG. As in the last section, the key reason for this is that computations involving SLCFG only require the consideration of contiguous strings.

It should be noted that in the special case of a fully bracketed training corpus, the parameters of an SCFG can be estimated in linear time [6, 10]. It is an open question whether this can be done for SLCFG. However, it should be straightforward to design an $O(n^2)$-time training algorithm for SLCFG given a fully bracketed corpus.

6 Conclusion

The preceding sections present stochastic lexicalized context-free grammar (SLCFG). SLCFG combines the processing speed of SCFG with the much greater ability of SLTAG to capture distributional information about words. As such, SLCFG has the potential of being a very useful tool for natural language processing tasks where statistical assessment/prediction is required.

References

[1] T. Booth. Probabilistic representation of formal languages. In *Tenth Annual IEEE Symposium on Switching and Automata Theory*, October 1969.

[2] F. Jelinek. Self-organized language modeling for speech recognition. In Alex Waibel and Kai-Fu Lee, editors, *Readings in speech recognition*. Morgan Kaufmann, San Mateo, California, 1990. Also in IBM Research Report (1985).

[3] F. Jelinek, J. D. Lafferty, and R. L. Mercer. Basic methods of probabilistic context free grammars. Technical Report RC 16374 (72684), IBM, Yorktown Heights, NY, 1990.

[4] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars and lexicalized grammars. In Maurice Nivat and Andreas Podelski, editors, *Tree Automata and Languages*. Elsevier Science, 1992.

[5] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

[6] Fernando Pereira and Yves Schabes. Inside-outside reestimation from partially bracketed corpora. In 20^{th} *Meeting of the Association for Computational Linguistics (ACL'92)*, Newark, Delaware, 1992.

[7] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[8] Philip Resnik. Probabilistic tree-adjoining grammars as a framework for statistical natural language processing. In *Proceedings of the* 14^{th} *International Conference on Computational Linguistics (COLING'92)*, 1992.

[9] Yves Schabes. Stochastic lexicalized tree-adjoining grammars. In *Proceedings of the* 14^{th} *International Conference on Computational Linguistics (COLING'92)*, 1992.

[10] Yves Schabes, Michael Roth, and Randy Osborne. Parsing the Wall Street Journal with the inside-outside algorithm. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics (EACL'93)*, Utrecht, the Netherlands, April 1993.

[11] Yves Schabes and Stuart Shieber. An alternative conception of tree-adjoining derivation. In 20^{th} *Meeting of the Association for Computational Linguistics (ACL'92)*, 1992.

[12] Yves Schabes and Richard C. Waters. Lexicalized context-free grammar: A cubic-time parsable formalism that strongly lexicalizes context-free grammar. Technical Report 93-04, Mitsubishi Electric Research Labs, 201 Broadway. Cambridge MA 02139, 1993.

[13] Yves Schabes and Richard C. Waters. Lexicalized context-free grammars. In 21^{st} *Meeting of the Association for Computational Linguistics (ACL'93)*, pages 121–129, Columbus, Ohio, June 1993.

[14] C. E. Shannon. Prediction and entropy of printed english. *The Bell System Technical Journal*, 30:50–64, 1951.

[15] J. W. Thatcher. Characterizing derivations trees of context free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 5:365–396, 1971.

Predictive Head-Corner Chart Parsing

Klaas Sikkel — Rieks op den Akker

Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE ENSCHEDE
email: {sikkel,infrieks}@cs.utwente.nl

Abstract

Head-Corner (HC) parsing has come up in computational linguistics a few years ago, motivated by linguistic arguments. This idea is a heuristic, rather than a fail-safe principle, hence it is relevant indeed to consider the worst-case behaviour of the HC parser. We define a novel predictive head-corner chart parser of cubic time complexity.

We start with a left-corner (LC) chart parser, which is easier to understand. Subsequently, the LC chart parser is generalized to an HC chart parser. It is briefly sketched how the parser can be enhanced with feature structures.

1 Introduction

"Our Latin teachers were apparently right", Martin Kay remarks in (Kay, 1989). "You should start [parsing] with the main verb. This will tell you what kinds of subjects and objects to look for and what cases they will be in. When you come to look for these, you should also start by trying to find the main word, because this will tell you most about what else to look for".

Head-driven or *head-corner* parsing has been addressed in several papers. (Proudian and Pollard, 1985; Kay 1989; Satta and Stock 1989; van Noord 1991; Bouma and van Noord 1993). As the head-driven approach is a heuristic, rather than a fail-safe principle, it is important to pay attention to the worst-case behaviour. This is best taken care of in a tabular approach like the bottom-up head-driven parser by Satta and Stock. We enhance the tabular head-driven parser with top-down prediction.

The algorithmic details of the head-corner parser are not easy. Therefore we will make some effort to convey the intuition behind the parser. To that end, we first define a *left-corner* chart parser in Section 3 and afterwards generalize this to a head-corner parser in 4. A complexity analysis is given in 5. We sketch extension with feature structures in 6 and briefly discuss related approaches in 7.

2 Chart parsing

Chart parsing, first introduced in (Kay 1980), is a well-known parsing technique in computational linguistics. We will present a conventional Earley chart parser in a slightly unconventional way. Thus the reader who is familiar with the Earley parser will get a feeling for the notation used in this article.

We use the following notational conventions. Nonterminals are denoted by $A, B, \ldots \in N$; terminals by $a, b, \ldots \in \Sigma$. We write V for $N \cup \Sigma$ with $X, Y, \ldots$ as typical elements. Strings in V^* are denoted by $\alpha, \beta, \ldots$. A context-free grammar G is a 4-tuple (N, Σ, P, S), with P a set of productions and S the start symbol. The sentence to be parsed is denoted $a_1 \ldots a_n$. We make extensive use of *place markers* $i, j, k \ldots$, indicating positions in the sentence. The symbol a_i is located between positions $i-1$ and i.

A chart parser is characterized by a *domain of items*, that can be added to the chart by the parser and some *operators* that specify how combinations of items on the chart can lead to recog-

nition of other items. Furthermore, there is an *initial chart* and *initial agenda*.

At each step some current item is selected from the agenda, and moved to the chart. If the chart contains items that, in combination with the current item, allow recognition of other items not yet present on the chart or on the agenda, these are added to the agenda. This continues until the agenda is empty.

A context-free chart parser does not really construct parse trees. But a representation of all parse trees can easily be obtained when items in the chart are annotated with pointers to the items that caused their recognition. Various forms of sophistication can be added by structuring the chart or by providing a strategy to select the next item from the agenda.

The Earley chart parser uses two types of items:

$[A \to \alpha\bullet\beta, i, j]$: *Earley items* (for $A \to \alpha\beta \in P$ and $0 \leq i \leq j \leq n$),
$[a, j-1, j]$: *terminal items* representing a_j $(1 \leq j \leq n)$.

An Earley item $[A \to \alpha\bullet\beta, i, j]$ is to be recognized by the chart parser iff

$\alpha \Rightarrow^* a_{i+1} \ldots a_j$, and
$S \Rightarrow^* a_1 \ldots a_i A\gamma$ for some $\gamma \in V^*$.

The initial chart contains the terminal items representing the string. When the j-th word belongs to different categories, say a and b, then both items $[a, j-1, j]$ and $[b, j-1, j]$ are present in the initial chart. The initial agenda contains items $[S \to \bullet\gamma, 0, 0]$ for all productions $S \to \gamma \in P$. The following operators are defined for the Earley chart parser:

predict: for $B \to \gamma \in P$:

$$[A \to \alpha\bullet B\beta, i, j] \vdash [B \to \bullet\gamma, j, j];$$

scan:

$$[A \to \alpha\bullet b\beta, i, j], [b, j, j+1] \vdash [A \to \alpha b\bullet\beta, i, j+1];$$

complete:

$$[A \to \alpha\bullet B\beta, i, j], [B \to \gamma\bullet, j, k] \vdash [A \to \alpha B\bullet\beta, i, k].$$

The turnstyle ($\vdash$) notation is a convenient shorthand, meaning that the left-hand side items licence the recognition of the right-hand side item. As a running example we will use the sentence `the cat caught a mouse`, represented by the lexical categories

$$*det\ *n\ *v\ *det\ *n.$$

The example grammar is

$$S \to NP\ VP,$$
$$NP \to *det\ *n,$$
$$VP \to *v\ NP.$$

Due to lack of ambiguity, the example will nicely illustrate the difference between the chart parsers that are presented in this paper. Initially, the chart contains $\{[0, *det, 1], [1, *n, 2], [2, *v, 3], [3, *det, 4], [4, *n, 5]\}$ and the agenda is $\{[S \to \bullet NP\ VP, 0, 0]\}$. In Figure 1 the completed chart is shown (excluding terminal items), annotated with how the items were recognized. The items can be seen as *the trace of a left-to-right walk through the parse tree*. This walk is shown in Figure 2 annotated with the item numbers. (In the general case things are rather more complicated. There could be different parse trees and also partial left-to-right walks of valid prefixes than can't be extended to a parse. All trese traces are interlaced and may partly coincide.)

3 Left-corner chart parsing

We will now specify a Left-Corner (LC) parser as a chart parser, that is to be generalized to a head-corner parser in the next section. The deterministic LC algorithm originally stems from (Rosenkrantz and Lewis, 1970). We describe a generalized LC parser[1] for context-free grammars

[1]The term "Generalized LC" has been introduced in (Demers, 1977) for a rather different concept. He generalized the notion of Left Corner, deriving a framework that describes a class of parsers and associated grammars ranging from LL(k) via LC(k) to LR(k). We generalize the LC(0) parser by dropping the restriction that the grammar be LC(0). The nondeterminism is handled by a dynamic programming technique, as in Generalized LR parsing (Tomita, 1985). Note

#	item	recognized by
0	$[S \rightarrow \bullet NP\ VP, 0, 0]$	initial
1	$[NP \rightarrow \bullet *det\ *n, 0, 0]$	*predict*(0)
2	$[NP \rightarrow *det\bullet\ *n, 0, 1]$	*scan*(1)
3	$[NP \rightarrow *det\ *n\bullet, 0, 2]$	*scan*(2)
4	$[S \rightarrow NP\bullet VP, 0, 2]$	*compl*(0,3)
5	$[VP \rightarrow \bullet *v\ NP, 2, 2]$	*predict*(4)
6	$[VP \rightarrow *v\bullet NP, 2, 3]$	*scan*(5)
7	$[NP \rightarrow \bullet *det\ *n, 3, 3]$	*predict*(6)
8	$[NP \rightarrow *det\bullet\ *n, 3, 4]$	*scan*(7)
9	$[NP \rightarrow *det\ *n\bullet, 3, 5]$	*scan*(8)
10	$[VP \rightarrow *v\ NP\bullet, 2, 5]$	*complete*(6,9)
11	$[S \rightarrow NP\ VP\bullet, 0, 5]$	*complete*(4,10)

Figure 1: The final Earley chart

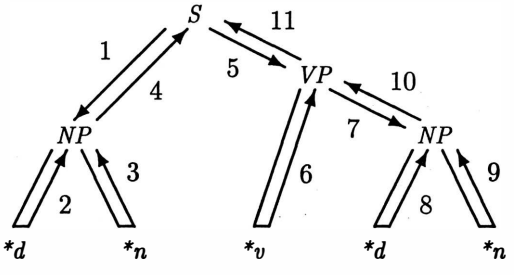


Figure 2: The Earley tree walk

that is similar to, but subtly different from the Earley parser. *Scan* and *complete* operations remain unchanged, predicting is handled differently. We start with a *predict item* $[0, S]$ meaning that we are to look for a constituent S starting at position 0. We proceed bottom-up, starting from $[*det, 0, 1]$. We can "climb up" from $*det$ to an NP because this moves us nearer to the goal S. Or, to put it more formally, because a leftmost derivation

$$S \Rightarrow^* NP\gamma \Rightarrow *det\ *n\gamma$$

exists, we may extend $[*det, 0, 1]$ to $[NP \rightarrow *det\bullet *n, 0, 1]$.

A few steps later we have recognized $[S \rightarrow NP\bullet VP, 0, 2]$. Here we set a *sub-goal*, represented by the predict item $[2, VP]$. In Figure 3 it is shown how the LC chart parser steps through a parse tree:

- Steps up correspond to a *scan* or *complete* as in the Earley case.
- Steps down to the leftmost child are skipped because these are implicitly encoded in the *transitive left-corner relation* that is encorporated in the parser.
- Steps down to *non-leftmost nonterminal* children correspond to setting a new subgoal.

that the semantic ambiguity of the noun phrase "Generalized LC parsing" duly reflects the syntactic ambiguity: we are concerned with [Generalized [Left-Corner Parsing]], whereas Demers discussed [[Generalized Left-Corner] parsing].

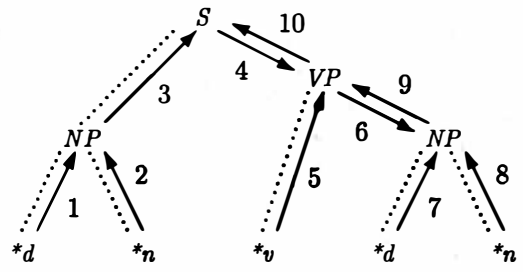


Figure 3: The left-corner tree walk

In Figure 4 the final chart is shown of the LC chart parser that will be formally defined next. Each item on the final chart corresponds to an arrow in the tree walk.

#	item	recognized by
0	$[0, S]$	initial
1	$[NP \rightarrow *det\bullet *n, 0, 1]$	$lc(i)$ (0)
2	$[NP \rightarrow *det\ *n\bullet, 0, 2]$	*scan* (1)
3	$[S \rightarrow NP\bullet\ VP, 0, 2]$	$lc(ii)$ (0,2)
4	$[2, VP]$	*predict* (3)
5	$[VP \rightarrow *v\bullet NP, 2, 3]$	$lc(i)$ (4)
6	$[3, NP]$	*predict* (5)
7	$[NP \rightarrow *det\bullet *n, 3, 4]$	$lc(i)$ (6)
8	$[NP \rightarrow *det\ *n\bullet, 3, 5]$	*scan* (7)
9	$[VP \rightarrow *v\ NP\bullet, 2, 5]$	*complete* (5,9)
10	$[S \rightarrow NP\ VP\bullet, 0, 5]$	*complete* (3,9)

Figure 4: The final LC chart

The intuition should be clear now, and we present the formal definition rather terse. The *left corner* of a production is the leftmost symbol in the right-hand side of that production (and ε for an empty production). We write $A >_\ell U$ if A has left corner $U \in (V \cup \{\varepsilon\})$ We write $>_\ell^*$ for the transitive closure of $>_\ell$. Hence, in the running example, we have $S >_\ell^* *det$.
The LC chart parser uses the following kinds of items:

$[i, A]$: predict items,
$[B \rightarrow \alpha\bullet\beta, i, j]$: Earley items (but only those with $\alpha \neq \varepsilon$ or $\alpha = \beta = \varepsilon$),
$[a, j-1, j]$: terminal items as usual.

The initial chart contains the terminal items representing the sentence, the agenda is initialized to $\{[0, S]\}$. The operators of the LC chart parser are defined as follows. We distinguish separate left-corner (lc) operators for left corners a, C, and ε.

$lc(i)$: for $A >_\ell^* B$, $B \rightarrow a\beta \in P$:

$$[i, A],\ [a, i, i+1] \vdash [B \rightarrow a\bullet\beta, i, i+1],$$

$lc(ii)$: for $A >_\ell^* B$, $B \rightarrow C\beta \in P$:

$$[i, A], [C \rightarrow \gamma\bullet, i, j] \vdash [B \rightarrow C\bullet\beta, i, j],$$

$lc(iii)$: for $A >_\ell^* B$, $B \rightarrow \varepsilon \in P$:

$$[i, A] \vdash [B \rightarrow \bullet, i, i];$$

predict:

$$[B \to \alpha\bullet C\beta, i, j] \vdash [j, C];$$

scan:

$$[B \to \alpha\bullet a\beta, i, j],\ [a, j, j+1] \vdash [B \to \alpha a\bullet\beta, i, j+1];$$

complete:

$$[B \to \alpha\bullet C\beta, i, j],\ [C \to \gamma\bullet, j, k] \vdash [B \to \alpha C\bullet\beta, i, k].$$

Thus we have characterized the LC chart parser by defining the initial chart and agenda and the operators. The reader may verify that these operators produce the chart shown in Figure 4 for our example sentence. For a proof of the correctness of the algorithm we refer to a more extensive technical report (Sikkel and op den Akker, 1992).

4 Head-Corner Chart Parsing

We introduce the head-corner chart parser by analogy to the left-corner parser. While the LC parser makes a left-to-right walk through a parse tree, the HC parser makes a *head-first* walk through a parse tree[2], as shown in Figure 5. With this very simple idea in mind we can fix the formal details.

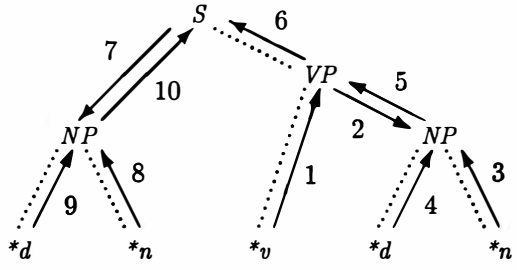


Figure 5: The head-corner tree walk

A *context-free head grammar* is a 5-tuple (N, Σ, P, S, r), with r a function that assigns a natural number to each production in P. Let $|p|$ denote the length of the right-hand side of p. Then r is constrained to $r(p) = 0$ for $|p| = 0$ and $1 \leq r(p) \leq |p|$ for $|p| > 0$. The *head* of a production p is the $r(p)$-th symbol of the right-hand side; an ε-production has head ε.

In a practical notation, we give a head grammar as a set of productions with the heads underlined. The head grammar H for our running example is given by

$$S \to NP\ \underline{VP},$$
$$VP \to \underline{*v}\ NP,$$
$$NP \to *det\ \underline{*n}.$$

The relation $>_h$ is defined by $A >_h U$ if there is a production $p = A \to \alpha \in P$ with U the head of p; the transitive and reflexive closure of $>_h$ is denoted $>_h^*$. In the running example it holds that $S >_h^* *v$. For the head-corner parser we distinguish the following kinds of items:

$[l, r, A]$: *predict items*
$[B \to \alpha\bullet\beta\bullet\gamma, i, j]$: *double dotted* (DD) *items*,
$[a, j-1, j]$: terminal items.

A predict item $[l, r, A]$ will be recognized if a constituent A is being looked for that must be located somewhere between l and r. Such a constituent should either stretch from l to some j (if we are working to the right from the head of some production) or from r downto some j (if we are

[2]In the general case, to be precise, the HC parser makes interlacing and partially coinciding walks through all full parse trees, as well as "valid infixes" starting from a feasible lexical head of the sentence.

working to the left from the head of some production), with $l \leq j \leq r$. A double dotted item $[B \rightarrow \alpha\bullet\beta\bullet\gamma, i, j]$ is recognized if there is a goal $[l, r, A]$ such that with $A >_h^* B$, $l \leq i$, $j \leq r$ and $\beta \Rightarrow^* a_{i+1} \ldots a_j$ has been established. In (Satta and Stock, 1992) it is remarked that, similar to the "valid prefix property" of the Earley and LC parser, our predictive head-corner parser satisfies a "valid infix property". As it is hard enough to get an understanding of what is going on, we will not concentrate on the mathematical details.

The initial chart contains the terminal items that represent the string as usual; the agenda is initialized with a single predict item $[0, n, S]$. If the string is correct, we will be able to derive an item $[S \rightarrow \bullet\gamma\bullet, 0, n]$. The operators, are defined as follows. We distinguish three different head-corner (hc) operators for heads b, C and ε.

$hc(i)$: for $A >_h^* B$, $B \rightarrow \alpha\underline{b}\gamma \in P$, $l < j \leq r$:

$$[l, r, A],\ [b, j-1, j] \vdash [B \rightarrow \alpha\bullet b\bullet\gamma, j-1, j],$$

$hc(ii)$: for $A >_h^* B$, $B \rightarrow \alpha\underline{C}\gamma \in P$, $l \leq i \leq j \leq r$:

$$[l, r, A],\ [C \rightarrow \bullet\delta\bullet, i, j] \vdash [B \rightarrow \alpha\bullet C\bullet\gamma, i, j],$$

$hc(iii)$: for $A >_h^* B$, $B \rightarrow \varepsilon \in P$, $l \leq j \leq r$:

$$[l, r, A] \vdash [B \rightarrow \bullet\bullet, j, j];$$

predict: for $A >_h^* B$, $l \leq i \leq j \leq r$:

$$[l, r, A],\ [B \rightarrow \alpha C\bullet\beta\bullet\gamma, i, j] \vdash [l, i, C],$$

$$[l, r, A],\ [B \rightarrow \alpha\bullet\beta\bullet C\gamma, i, j] \vdash [j, r, C];$$

scan: for $A >_h^* B$, $l < j \leq k \leq r$:

$$[l, r, A],\ [a, j-1, j],\ [B \rightarrow \alpha a\bullet\beta\bullet\gamma, j, k] \vdash [B \rightarrow \alpha\bullet a\beta\bullet\gamma, j-1, k],$$

$$[l, r, A],\ [B \rightarrow \alpha\bullet\beta\bullet a\gamma, i, j-1],\ [a, j-1, j] \vdash [B \rightarrow \alpha\bullet\beta a\bullet\gamma, i, j];$$

complete: for $A >_h^* B$, $l \leq i \leq j \leq k \leq r$:

$$[l, r, A],\ [C \rightarrow \bullet\delta\bullet, i, j],\ [B \rightarrow \alpha C\bullet\beta\bullet\gamma, j, k] \vdash [B \rightarrow \alpha\bullet C\beta\bullet\gamma, i, k],$$

$$[l, r, A],\ [B \rightarrow \alpha\bullet\beta\bullet C\gamma, i, j],\ [C \rightarrow \bullet\delta\bullet, j, k] \vdash [B \rightarrow \alpha\bullet\beta C\bullet\gamma, i, k].$$

Head-corner analysis of the example sentence illustrates how the chart parser may jump up and down the sentence. The completed HC chart is shown in Figure 6. Each item corresponds to a step in the head-corner tree walk for the parse of our example sentence, cf. Figure 5.

#	item	recognized by
0	$[0, 5, S]$	*initial*
1	$[VP \rightarrow \bullet *v\bullet NP, 2, 3]$	$hc(i)$ (0)
2	$[3, 5, NP]$	*predict*(1)
3	$[NP \rightarrow *det\bullet *n\bullet, 4, 5]$	$hc(i)$ (2)
4	$[NP \rightarrow \bullet *det\ *n\bullet, 3, 5]$	*scan* (3)
5	$[VP \rightarrow \bullet *v\ NP\bullet, 2, 5]$	*complete* (1,4)
6	$[S \rightarrow NP\bullet VP\bullet, 2, 5]$	$hc(ii)$ (0,5)
7	$[0, 2, NP]$	*predict*(6)
8	$[NP \rightarrow *det\bullet *n\bullet, 1, 2]$	$hc(i)$ (7)
9	$[NP \rightarrow \bullet *det\ *n\bullet, 0, 2]$	*scan* (8)
10	$[S \rightarrow \bullet NP\ VP\bullet, 0, 5]$	*complete* (6,9)

Figure 6: The HC chart

5 Complexity analysis and further optimizations

The number of items that can be recognized now is $O(n^2)$, but the work involved for an arbitrary current item is more than linear. The most problematic operation is *complete* (with *scan* as a special sub-case) with 5 place markers involved.

Complete can be reduced to 3 place markers with some special extra bookkeeping. As a consequence, the number of place markers involved in *scan* and *predict* will drop from 4 to 3. We keep a *goal table*, in the form of a CYK table, for storage of the predicted goals. Whenever an item $[l, r, A]$ is predicted, we write an A in goal table entry (l, r). Furthermore, we write an A in every entry (i, j) with $l \leq i \leq j \leq r$ in which no A is present. A typical case is presented in Figure 7. A goal $[0, 5, A]$ is to be added, the entry (0,5) is indicated * in Figure 7(a). One adds A symbols column by column, stopping each time when an A is found. In Figure 7(b) a * indicates the entries where an A is written and + indicates the entries that were inspected but already contained an A. During the course of the algorithm only $O(n^2)$ A symbols are written, per A only $O(n)$ entries are inspected that already did contain A.

```
      0 1 2 3 4 5 6 (j)
    0 A A A . . * .
    1   A A . . . .
    2     A . . . .
    3       . . . .
    4         A A A
    5           A A
(i) 6             A
```

(a)

```
      0 1 2 3 4 5 6 (j)
    0 A A + * * * .
    1   A A * * * .
    2     A * * * .
    3       * * * .
    4         + + A
    5           A A
(i) 6             A
```

(b)

Figure 7: Adding a goal to the goal table

Using the goal table, the place markers l, r in the *complete* can be replaced by i, k. Using the goal table as explained above one can straightforwardly verify that the overall complexity in the length of the sentence is $O(n^2)$ space and $O(n^3)$ time as usual, assuming that the chart is structured likewise as a triangular matrix. When the size of the grammar is taken into account the analysis gets somewhat more complicated. For the sake of brevity we only state the results. For an optimal worst-case complexity we have to make a small change to the implementation and represent fully completed DD items $[A \rightarrow \bullet\beta\bullet, i, j]$ by CYK items $[A, i, j]$. An Earley chart parser — using the same optimization — has a time complexity $O(|G|n^3)$, with $|G|$ the number of productions multiplied by the average length of a production. *Without prediction* the HC parser has a time complexity of $O(|G|rn^3)$, with r the size of the longest production. This extra factor r is because we use double dotted, rather than single dotted items. Prediction usually speeds up a parser but may slow it down in pathological cases. In the worst case it adds a factor $|N|$, the number of nonterminal symbols, yielding a time complexity of $O(|N||G|rn^3)$. The space complexity is $O(|G|rn^2)$ for the chart and agenda, and $O(|N|^2 \times |N||\Sigma|)$ for storage of the $>_h^*$ relation in tabular form.

The obtained worst-case complexity is optimal, in the sense that all complexity factors are properly accounted for (i.e., the factors r and $|N|$ in addition to an optimal Earley parser are evidently necessary). Yet, on a practical level, a large percentage of computing time can be saved by adding some more sophistication to the algorithm. We will not formally introduce an optimized algorithm, as the definitions grow rather complicated, but simply state some principles that can be implemented straightforwardly.

- When an item $[A \rightarrow \alpha\bullet B\bullet\gamma, i, j]$ has been recognized by the head-corner rule, with $\alpha \neq \varepsilon \neq \gamma$, it should be expanded *either* to $[A \rightarrow \bullet\alpha B\bullet\gamma, h, j]$ *or* to $[A \rightarrow \alpha\bullet B\gamma\bullet, i, k]$ but not both; from either one a completed item $[A \rightarrow \bullet\alpha B\gamma\bullet, h, k]$ can be obtained. This idea is taken from (Satta and Stock 1989).

- A predicted item should fit to the left, fit to the right, or both. This can be expressed by using predict items of the form $[= l, = r, A]$, $[\geq l, = r, A]$ and $[= l, \leq r, A]$ with the obvious interpretation. For example, if $A >_h^* X$ and X occurs only at the left (i.e., if $A \Rightarrow^* X\beta$) then $[X, i, j]$ fits to $[= l, \leq r, A]$ only if $i = l$. The head-corner operator can be adapted accordingly.

If the grammar is limited to Chomsky Normal Form, the first saving doesn't apply but the second is more effective. Further optimizations are possible, but beyond the scope of this article.

6 Extension with feature structures

Feature information according to any unification based grammar can be added to recognized items. For bottom-up composition of constituents this is straightforward. Using feature information for top-down prediction is more subtle, as we can only use so-called *transitive features*. A feature is transitive if every constituent shares the feature with its head. Hence, a constituent transitively shares such a feature with its lexical head. This is important, because sub-goals are parsed from the lexical head upwards. If, for example, a *VP* has been found with *agreement* 3sg (third person singular), then the *NP* that is set as a goal must also have *agreement* 3sg. Because *agreement* is a transitive feature, only a noun with *agreement* 3sg can be the lexical head.

Practically this works as follows. Double dotted items $[B \rightarrow \alpha \bullet \beta \bullet \gamma, i, j]$ are replaced by items $[l, r, A; B \rightarrow \alpha \bullet \beta \bullet \gamma, i, j]$, with A the predicted goal symbol. The transitive features of A are shared with B. If we have an item $[l, r, A; A \rightarrow \bullet\alpha\bullet, l, j]$ we can unify the predicted and parsed A if the nontransitive features match as well.

7 Related approaches

The left-corner chart parser of Section 3, although rather different in style of presentation, is closely related to the predictive chart parsing framework introduced in (Kay, 1980). First ideas of generalized LC parsing, although not under that name, can be traced back to (Pratt, 1975). A left-corner style parser in Prolog was presented in (Matsumoto et al, 1983). This BUP parser is limited to acyclic, ε-free grammars. In (Nederhof, 1992)a generalized LC parser is defined by analogy to Tomita's generalized LR parser (Tomita, 1985). Nederhof's parser is more efficient than our LC chart parser, but it doesn't generalize to HC.

The head-driven parser in (Satta and Stock, 1989) is similar to ours, but does not make use of prediction. The use of a head-driven approach to enhance the efficiency of prediction was first suggested in (Kay, 1989).

The context-free head grammars in Section 4 should not be confused with Head Grammars as introduced in (Pollard, 1984). that handle discontinuous constituents by means of "head wrapping". (Van Noord, 1991) describes a Prolog implementation of a head-corner parser for languages with discontinuous constituents.

Bouma and van Noord have experimented with various parsing strategies for unification grammars (Bouma and van Noord, 1993) and conclude that for important classes of grammars it is fruitful to apply parsing strategies that are sensitive to the linguistic notion of a head.

8 Conclusions and future work

We have given a formal treatment of a predictive head-corner parser. The item-based description of (predictive) chart parsers is a useful formalism for such a formal treatment. This is exemplified by the fact that we cover grammars with ε-productions with hardly any additional effort, while these are usually left out for the sake of simplicity. Enhancing a head-corner chart parser with prediction is new.

It cannot be stated in general that the head-corner approach is more efficient than the (generalized) left-corner approach or other parsers. It is indeed a heuristic, that can be expected to be effective when most of the feature information of a constituent is located in the head. Hence, *because* it is a method based on a heuristic, rather than a fail-safe principle, it is important to consider what happens if the heuristic doesn't pay off. Therefore we have made some effort to make sure that the worst-case behaviour conforms to the usual complexity bounds for context-free parsing algorithms: $O(n^3)$ time and $O(n^2)$ space.

We have indicated how the algorithm can be extended with feature information. An implementation of a head-corner chart parser for PATR-like unification grammars is (nearly) finished. It is currently being tested on a natural language grammar developed for a knowledge representation research project at our institute. We intend to make an extensive comparison of

the efficiency of the head-corner and left-corner parser.

Acknowledgements

We are grateful to Anton Nijholt, Mark-Jan Nederhof, Gertjan van Noord and Giorgio Satta for constructive comments on earlier drafts and Margriet Verlinden for implementing the HC chart parser for PATR.

References

Bouma, G. and G. van Noord (1993). Head-driven Parsing for Lexicalist Grammars: Experimental Results. *6th European Chapter of the ACL*, 71–80.

Demers, A.J. (1977). Generalized Left Corner Parsing. *4th ACM Symp. on Principles of Prog. Lang.*, 170–182.

Kay, M. (1980). Algorithm Schemata and Data Structures in Syntactic Processing. Report CSL-80-12, Xerox PARC, Palo Alto, Ca. Reprinted in: GROSZ, B.J. et al. (Eds.), *Readings in Natural Language Processing*, Morgan Kaufmann, Los Altos, Ca. (1982).

Kay, M. (1989). Head Driven Parsing. *Int. Workshop on Parsing Technologies (IWPT'89)*, 52–62.

Matsumoto, Y., H. Tanaka, H. Hirakawa, H. Miyoshi and H. Yasukawa (1983). BUP: a bottom-up parser embedded in Prolog. *New Generation Computing* **1**, 145–158.

Nederhof, M.-J. (1993). Generalized Left-Corner Parsing. *6th European Chapter of the ACL*, 305–314.

Noord, G. van (1991). Head Corner Parsing for Discontinuous Constituency. *29th Ann. Meeting of the ACL*, 114–121.

Pollard, C. (1984). *Generalized Context-Free Grammars, Head Grammars and Natural Languages*. Ph.D. Thesis, Dept. of Linguistics, Stanford University.

Pratt, V.R. (1975). LINGOL — A Progress Report. *4th Int. Joint Conf. on Artificial Intelligence (IJCAI'75)*, 422–428.

Proudian, D. and C. Pollard (1985). Parsing head-driven phrase structure grammar. *23th Ann. Meeting of the ACL*, 167–171.

Rosenkrantz, D.J. and P.M. Lewis (1970). Deterministic Left Corner Parsing. *11th Ann. Symp. on Switching and Automata Theory*, 139–152.

Satta, G. and O. Stock (1989). Head-Driven Bidirectional Parsing: A Tabular Method. *Int. Workshop on Parsing Technologies (IWPT'89)*, 43–51.

Satta, G. and O. Stock (1992). BiDirectional Context-Free Grammar Parsing for Natural Language Processing. Technical Report IRCS-92-13, Institute for Research in Cognitive Science, University of Pennsylvania, Philadelphia.
To appear in *J. of Artificial Intelligence.*

Sikkel, K. and R. op den Akker (1992). Left-Corner and Head-Corner Chart Parsing. Memoranda Informatica 92-55, University of Twente, Enschede, the Netherlands.

Tomita, M. (1985). *Efficient Parsing for Natural Language*. Kluwer, Boston, Mass.

Parsing English with a Link Grammar

Daniel D. Sleator* and Davy Temperley†

*School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
email: sleator@cs.cmu.edu

†Music Department
Columbia University
New York, NY 10027
email: dt3@cunixa.cc.columbia.edu

Abstract

We define a new formal grammatical system called a *link grammar*. A sequence of words is in the language of a link grammar if there is a way to draw *links* between words in such a way that (1) the local requirements of each word are satisfied, (2) the links do not cross, and (3) the words form a connected graph. We have encoded English grammar into such a system, and written a program (based on new algorithms) for efficiently parsing with a link grammar. The formalism is lexical and makes no explicit use of constituents and categories. The breadth of English phenomena that our system handles is quite large. A number of sophisticated and new techniques were used to allow efficient parsing of this very complex grammar. Our program is written in C, and the entire system may be obtained via anonymous ftp. Several other researchers have begun to use link grammars in their own research.

1 Introduction

Most sentences of most natural languages have the property that if arcs are drawn connecting each pair of words that relate to each other, then the arcs will not cross [10, p. 36]. This well-known phenomenon, which we call *planarity*, is the basis of *link grammars* our new formal language system.

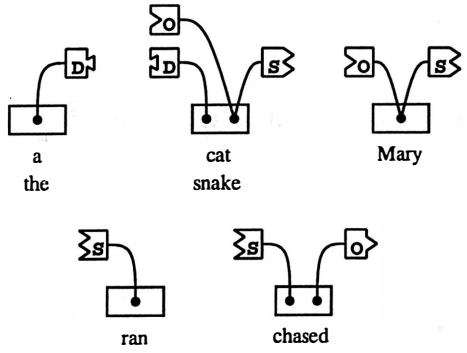
A link grammar consists of a set of *words* (the terminal symbols of the grammar), each of which has a *linking requirement*. A sequence of words is a *sentence* of the language defined by the grammar if there exists a way to draw *links* among the words so as to satisfy the following conditions:

Planarity: The links do not cross (when drawn above the words).

Connectivity: The links suffice to connect all the words of the sequence together.

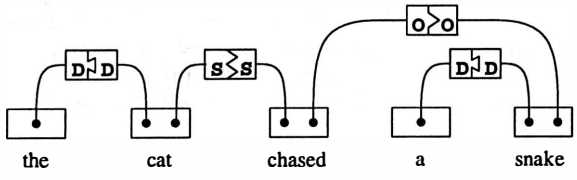
Satisfaction: The links satisfy the linking requirements of each word in the sequence.

The linking requirements of each word are contained in a *dictionary*. To illustrate the linking requirements, the following diagram shows a simple dictionary for the words *a*, *the*, *cat*, *snake*, *Mary*, *ran*, and *chased*. The linking requirement of each word is represented by the diagram above the word.

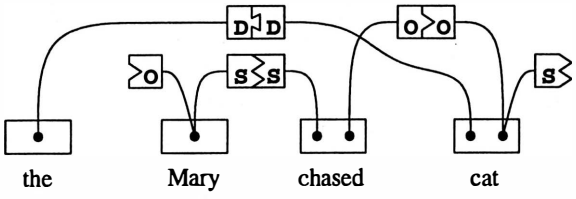


Each of the intricately shaped labeled boxes is a *connector*. A connector is satisfied by matching it to a compatible connector (one with the appropriate shape, facing in the opposite direction). Exactly one of the connectors attached to a given black dot must be satisfied (the others, if any, must not be used). Thus, *cat* requires a D connector to its left, and either an O connector to its left or a S connector to its right. Plugging a pair of connectors together corresponds to drawing a link between that pair of words.

The following diagram shows how the linking requirements are satisfied in the sentence *The cat chased a snake.*

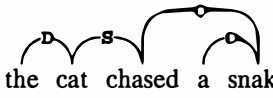


(The unused connectors have been suppressed here.) It is easy to see that *Mary chased the cat*, and *the cat ran* are also sentences of this grammar. The sequence of words: *the Mary chased cat* is not in this language. Any attempt to satisfy the linking requirements leads to a violation of one of the three rules. Here is one attempt:



Similarly *ran Mary*, and *cat ran chased* are not part of this language.

A set of links that prove that a sequence of words is in the language of a link grammar is called a *linkage*. From now on we'll use simpler diagrams to illustrate linkages. Here is the simplified form of the diagram showing that *the cat chased a snake* is part of this language:



We have a succinct, computer-readable notation for expressing the dictionary of linking requirements. The following dictionary encodes the linking requirements of the previous example.

words	formula
a the	D+
snake cat	D- & (O- or S+)
Mary	O- or S+
ran	S-
chased	S- & O+

The linking requirement for each word is expressed as a formula involving the operators &, and or, parentheses, and connector names. The + or - suffix on a connector name indicates the direction (relative to the word being defined) in which the matching connector (if any) must lie. The & of two formulas is satisfied by satisfying both the formulas. The or of two formulas requires that exactly one of its formulas be satisfied. The order of the arguments of an & operator is significant. The farther left a connector is in the expression, the nearer the word to which it connects must be. Thus, when using *cat* as an object, its determiner (to which it is connected with its D- connector) must be closer than the verb (to which it is connected with its O- connector).

We can roughly divide our work on link grammars into three parts: the link grammar formalism and its properties, the construction of a wide-coverage link grammar for English, and efficient algorithms and techniques for parsing link grammars. We now touch briefly on all three of these aspects.

Link grammars are a new and elegant context-free grammatical formalism[1][2], and have a unique combination of useful properties:

[1] Link grammars resemble dependency grammars and categorial grammars. There are also many significant differences. Some light is shed on the relationship in section 6.

[2] The proof of the context-freeness of link grammars is not included in this paper, but appears in our technical

1. In a link grammar each word of the lexicon is given a definition describing how it can be used in a sentence. The grammar is distributed among the words. Such a system is said to be *lexical*. This has several important advantages. It makes it easier to construct a large grammar, because a change in the definition of a word only affects the grammaticality of sentences involving that word. The grammar can easily be constructed incrementally. Furthermore, expressing the grammar of the irregular verbs of English is easy – there's a separate definition for each word.

   Another nice feature of a lexical system is that it allows the construction of useful probabilistic language models. This has led researchers to construct lexical versions of other grammatical systems, such as tree-adjoining grammars [13]. Lafferty and the present authors have also constructed such a probabilistic model for link grammars [11].

2. Unlike a phrase structure grammar, after parsing a sentence with a link grammar words that are associated semantically and syntactically are directly linked. This makes it easy to enforce agreement, and to gather statistical information about the relationships between words.

3. In English, whether or not a noun needs a determiner is independent of whether it is used as a subject, an object, or even if it is part of a prepositional phrase. The algebraic notation we developed for expressing a link grammar takes advantage of this orthogonality. Any lexical grammatical system, if it is to be used by a human, must have such a capability. In our current on-line dictionary the word *cat* can be used in 369 different ways, and for *time* this number is 1689. A compact link grammar formula captures this large number of possibilities, and can easily be written and comprehended by a human being.

4. Another interesting property of link grammars is that they have no explicit notion of constituents or categories. In most sentences parsed with our dictionaries, constituents can be seen to emerge as contiguous connected collections of words attached to the rest of the sentence by a particular type of link. For example in the dictionary above, `S` links always attach a noun phrase (the connected collection of words at the left end of the link) to a verb (on the right end of the link). `O` links work in a similar fashion. In these cases the links of a sentence can be viewed as an alternative way of specifying the constituent structure of the sentence. On the other hand, this is not the way we think about link grammars, and we see no advantage in taking that perspective.

Our second result is the construction of a link grammar dictionary for English. The goal we set for ourselves was to make a link grammar that can distinguish, as accurately as possible, syntactically correct English sentences from incorrect ones. We chose a formal or newspaper-style English as our model. The result is a link grammar of roughly eight hundred definitions (formulas) and 25000 words that captures many phenomena of English grammar. It handles: noun-verb agreement, questions, imperatives, complex and irregular verbs, many types of nouns, past- or present-participles in noun phrases, commas, a variety of adjective types, prepositions, adverbs, relative clauses, possessives, coordinating conjunctions, unbounded dependencies, and many other things.

The third result described in this paper is a program for parsing with link grammars. The program reads in a dictionary (in a form very similar to the tables above) and will parse sentences according to the given grammar. The program does an exhaustive search – it finds every way of parsing the given sequence with the given link grammar. It is based on our own $O(n^3)$ algorithm (n is the number of words in the sentence to be parsed). The program also makes use of several very effective data structures and heuristics to speed up parsing. The program is comfortably fast – parsing typical newspaper sentences in a few seconds on a modern workstation.

Both our program (written in ANSI-C) and our dictionary are available via anonymous ftp

report [14]. Note that context-free systems can differ in many ways, including the ease with which the same grammar can be expressed, the efficiency with which the same grammar can be parsed, and the usefulness of the output of the parser for further processing.

through the internet.[3] Having the program available for experimentation may make it easier to understand this paper.

The organization of this paper

In section 2 we define link grammars more formally and explain the notation and terminology used throughout the rest of the paper. In section 3 we describe the workings of a small link grammar for English. Our $O(n^3)$ algorithm is described in section 4, and the data structures and heuristics that make it run fast are described in section 5. In section 6 we explain the relationship between link grammars, dependency syntax, and categorial grammars. We show how to automatically construct a link grammar for a given categorial grammar. This construction allows our efficient parsing algorithms and heuristics to be applied to categorial grammars. Section 7 mentions several other research projects that are based on link grammars.

Space limitations prevent us from presenting details of a number of other aspects of our work. The following paragraphs mention a few of these. More details on all of these matters are contained in our technical report [14].

There are a number of common English phenomena that are not handled by our current system. Our technical report contains a list of these, along with the reason for this state of affairs. The reasons range from the fact that ours is a preliminary system to the fact that some phenomena simply do not fit well into the link grammar framework.

Coordinating conjunctions such as *and* pose a problem for link grammars. This is because in a sentence like *The dog chased and bit Mary* there should logically be links between both *dog* and *bit* and *chased* and *Mary*. Such links would cross. We have devised a scheme that handles the vast majority of uses of such conjunctions and incorporated it into our program. The existence of such a conjunction in a sentence modifies the grammar of the words in it. The same parsing algorithm is then used on the resulting modified grammar.

Certain other constructs are difficult to handle only using the basic link grammar framework. One example is the non-referential use of *it*: *It is likely that John will go* is correct, but *The cat is likely that John will go* is wrong. It is possible – but awkward – to distinguish between these with a link grammar. To deal with this (and a number of other phenomena), we extended the basic link grammar formalism with a *post-processor* that begins with a linkage, analyzes its structure, and determines if certain conditions are satisfied. This allows the system to correctly judge a number of subtle distinctions (including that mentioned here).

2 Notation and terminology

2.1 Meta-rules

The link grammar dictionary consists of a collection of entries, each of which defines the linking requirements of one or more words. These requirements are specified by means of a *formula* of *connectors* combined by the binary associative operators `&` and `or`. Presidence is specified by means of parentheses. Without loss of generality we may assume that a connector is simply a character string ending in `+` or `-`.

When a link connects to a word, it is associated with one of the connectors of the formula of that word, and it is said to *satisfy* that connector. No two links may satisfy the same connector. The connectors at opposite ends of a link must have names that *match*, and the one on the left must end in `+` and the one on the right must end in `-`. In basic link grammars, two connectors match if and only if their strings are the same (up to but not including the final `+` or `-`). A more general form of matching will be introduced later.

The connectors satisfied by the links must serve to satisfy the whole formula. We define the notion of satisfying a formula recursively. To satisfy the `&` of two formulas, both formulas must be satisfied. To satisfy the `or` of two formulas, one of the formulas must be satisfied, and *no* connectors of the other formula may be satisfied. It is sometimes convenient to use the empty formula ("`()`"), which is satisfied by being connected to no links.

A sequence of words is a *sentence* of the language defined by the grammar if there exists a

[3] The directory is `/usr/sleator/public` on the host `spade.pc.cs.cmu.edu` (128.2.209.226). Our technical reports [14, 11] are also available there.

way to draw links among the words so as to satisfy each word's formula, and the following *meta-rules*:

Planarity: The links are drawn above the sentence and do not cross.

Connectivity: The links suffice to connect all the words of the sequence together.

Ordering: When the connectors of a formula are traversed from left to right, the words to which they connect proceed from near to far. In other words, consider a word, and consider two links connecting that word to words to its left. The link connecting the nearer word (the shorter link) must satisfy a connector appearing to the left (in the formula) of that of the other word. Similarly, a link to the right must satisfy a connector to the left (in the formula) of a longer link to the right.

Exclusion: No two links may connect the same pair of words.

2.2 Disjunctive form

The use of formulas to specify a link grammar dictionary is convenient for creating natural language grammars, but it is cumbersome for mathematical analysis of link grammars, and in describing algorithms for parsing link grammars. We therefore introduce a different way of expressing a link grammar called *disjunctive form.*

In disjunctive form, each word of the grammar has a set of *disjuncts* associated with it. Each disjunct corresponds to one particular way of satisfying the requirements of a word. A disjunct consists of two ordered lists of connector names: the *left list* and the *right list.* The left list contains connectors that connect to the left of the current word (those connectors end in -), and the right list contains connectors that connect to the right of the current word. A disjunct will be denoted:

$$((L_1, L_2, \ldots, L_m)\ \ (R_n, R_{n-1}, \ldots, R_1))$$

Where $L_1, L_2, \ldots, L_m$ are the connectors that must connect to the left, and $R_1, R_2, \ldots, R_n$ are connectors that must connect to the right. The number of connectors in either list may be zero. The trailing + or - may be omitted from the connector names when using disjunctive form, since the direction is implicit in the form of the disjunct.

To satisfy the linking requirements of a word, one of its disjuncts must be satisfied (and no links may attach to any other disjunct). To satisfy a disjunct all of its connectors must be satisfied by appropriate links. The words to which $L_1, L_2, \ldots$ are linked are to the left of the current word, and are monotonically increasing in distance from the current word. The words to which $R_1, R_2, \ldots$ are linked are to the right of the current word, and are monotonically increasing in distance from the current word.

It is easy to see how to translate a link grammar in disjunctive form to one in standard form. This can be done simply by rewriting each disjunct as

$$(\ L_1\ \&\ L_2\ \&\ \cdots\ \&\ L_m\ \&\ R_1\ \&\ R_2\ \&\ \cdots\ \&\ R_n\ )$$

and combining all the disjuncts together with the `or` operator to make an appropriate formula.

It is also easy to translate a formula into a set of disjuncts. This is done by enumerating all ways that the formula can be satisfied. For example, the formula:

```
(A- or ()) & D- & (B+ or ()) & (O- or S+)
```

corresponds to the following eight disjuncts:

```
  ((A,D)   (S,B))
((A,D,O)   (B))
  ((A,D)   (S))
((A,D,O)   ())
    ((D)   (S,B))
  ((D,O)   (B))
    ((D)   (S))
  ((D,O)   ())
```

2.3 Our dictionary language

To streamline the difficult process of writing the dictionary, we have incorporated several other features to the dictionary language. Examples of all of these features can be found in section 3.

It is useful to consider connector matching rules that are more powerful than simply requiring the strings of the connectors to be identical. The most general matching rule is simply a table – part of the link grammar – that specifies all pairs of connectors that match. The resulting link grammar is still context-free.

In the dictionary presented later in this paper, and in our larger on-line dictionary, we use a matching rule that is slightly more sophisticated than simple string matching. We shall now describe this rule.

A connector name begins with one or more upper case letters followed by a sequence of lower case letters or *s. Each lower case letter (or *) is a *subscript*. To determine if two connectors match, delete the trailing + or -, and append an infinite sequence of *s to both connectors. The connectors match if and only if these two strings match under the proviso that * matches a lower case letter (or *).

For example, S matches both Sp and Ss, but Sp does not match Ss. Similarly, D*u, matches Dmu and Dm, but not Dmc. All four of these connectors match Dm.

The formula "((A- & B+) or ())" is satisfied either by using both A- and B+, or by using neither of them. Conceptually, then, the the expression "(A+ & B+)" is optional. Since this occurs frequently, we denote it with curly braces, as follows: {A+ & B+}.

It is useful to allow certain connectors to be able to connect to one or more links. This makes it easy, for example, to allow any number of adjectives to attach to a noun. We denote this by putting an "@" before the connector name, and call the result a *multi-connector*.

Our dictionaries consist of a sequence of *entries*, each of which is a list of words separated by spaces, followed by a colon, followed by the formula defining the words, followed by a semicolon.

3 An example

Perhaps the best way to understand how to write a link grammar for English is to study an example. The following dictionary does not cover the complete grammar of the words it contains, but it does handle a number of phenomena: verb-noun agreement, adjectives, questions, infinitives, prepositional phrases, and relative clauses.

```
the:        D+;
a:          Ds+;
John Mary:
  J- or O- or (({C- or CL-} & S+) or SI-);
dog cat park bone stick:
  {@A-} & Ds-
  & {@M+ or (C+ & Bs+)}
  & (J- or O- or ({C- or CL-} & Ss+) or SIs-);
dogs cats parks bones sticks:
  {@A-} & {Dm-}
  & {@M+ or (C+ & Bp+)}
  & (J- or O- or ({C- or CL-} & Sp+) or SIp-);
has:
  (SIs+ or Ss- or (Z- & B-))
  & (((B- or O+) & {@EV+}) or T+);
did:
  (SI+ & I+)
  or ((S- or (Z- & B-))
     & (((B- or O+) & {@EV+}) or I+));
can may will must:
  (SI+ or S- or (Z- & B-)) & I+;
is was:
  (Ss- or (Z- & Bs-) or SIs+)
  & (AI+ or O+ or B- or V+ or Mp+);
touch chase meet:
  (Sp- or (Z- & Bp-) or I-)
  & (O+ or B-) & {@EV+};
touches chases meets:
  (Ss- or (Z- & Bs-)) & (O+ or B-) & {@EV+};
touched chased met:
  (V- or M-
    or ((S- or (Z- & B- ) or T-) & (O+ or B-)))
  & {@EV+};
touching chasing meeting:
  (GI- or M-) & (O+ or B-) & {@EV+};
die arrive:
  (Sp- or (Z- & Bp-) or I-) & {@EV+};
dies arrives:
  (Ss- or (Z- & Bs-)) & {@EV+};
died arrived:
  (S- or (Z- & B-) or T-) & {@EV+};
dying arriving:
  (GI- or M-) & {@EV+};
with in by:
  J+ & (Mp- or EV-);
big black ugly:
  A+ or (AI- & {@EV+});
who:
  (C- & {Z+ or CL+}) or B+ or Ss+;
```

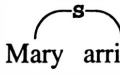
3.1 Some Simple Connectors

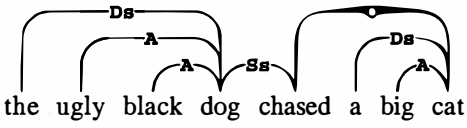
We develop an explanation of how this works in stages. Let's first restrict our attention to the following connectors: S, O, A, D. (Imagine the dictionary with all of the other connectors removed.) The S is used to connect a noun to its verb. The O connector is used to connect a verb to its object. The A connector is used to connect an adjective to its noun. The D is for connecting a determiner

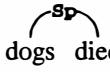
to its noun. Notice that this connector is omitted from proper nouns, is optional on plural nouns, and is mandatory on singular nouns. Also notice that the subscripts allow *the* to act as the determiner for both plural and singular nouns, but *a* can only work with the singular nouns. Similarly, the S connector is subscripted to ensure verb-noun agreement.

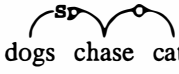
The ordering of the terms in these expressions is often important. For example, the fact that on nouns, the A- occurs to the left of the D- means that the adjective must be closer to the noun than the determiner.

Here are some judgements that can be rendered by what we have described so far:





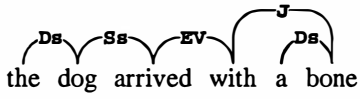


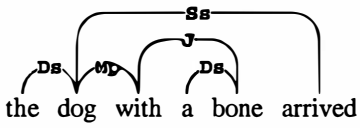


*a dog chase a cat
*black the dog died
*a/*the Mary chased the cat
*a dogs died
*dog died

3.2 Prepositions

The J, M and EV connectors allow prepositional phrases. The J connector connects a preposition to its object. Notice that in nouns, the J- is an alternative to the O-. This means that a noun cannot both be an object of a verb and of a preposition. The M connector is used when a prepositional phrase modifies a noun and the EV connector is used when a prepositional phrase modifies a verb. The following two examples illustrate this:

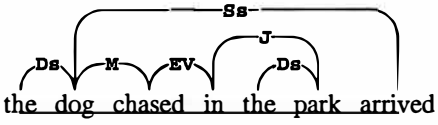


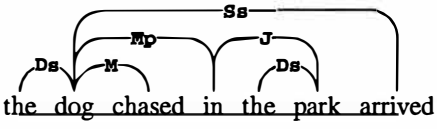


Notice that, as with A- connectors on nouns, a @ is used for M- connectors on nouns and EV- connectors on verbs, allowing multiple prepositional phrases, such as *John chased a dog in the park with a stick.*

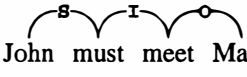
3.3 Participles

The M- connector on *chased* allows it to act as a participle phrase modifying a noun, as shown in these examples.





The I connector is used for infinitives, as in:

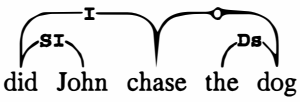


Notice that the I connector is an alternative to the S connector on plural verb forms. Thus we take advantage of the fact that plural verb forms are usually the same as infinitive forms, and include them both in a single dictionary entry.

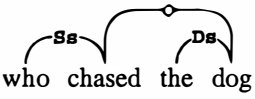
In a similar way, the T connector is used for past participles. Past participles have a T-; forms of the verb *have* have a T+. The GI connector is used for present participles. Present participles have a GI- connector; forms of the verb *be* have a GI+. The AI connector is used for predicative adjectives. Adjectives have a AI- connector; forms of *be* have a AI+ connector.

3.4 Questions

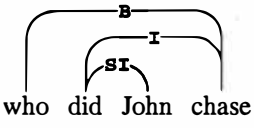
The SI connector is used for questions where there is subject-verb inversion. On nouns SI- is an alternative to S+, and on invertible verbs (is, has, did, must, etc.) SI+ is an alternative to S-. This allows



Wh- questions work in various different ways; only questions involving *who* will be discussed here. For subject-type questions, where *who* is substituting for the subject, *who* simply has an S+ connector. This allows



For object-type questions, where *who* is substituting for the object, the B connector is used. Transitive verbs have B- connectors as an alternative to their O+ connectors. *Who* has a B+ connector. This allows



The following incorrect sentences are rejected:

*Did John chase
*Who did John chase Mary
*John did Mary chase
*Chased John Mary

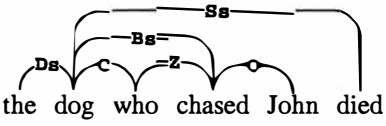
The following incorrect construction is accepted. In our on-line system, post-processing is used to eliminate this.

*Who John chased

3.5 Relative Clauses

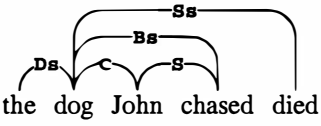
For subject-type relative clauses, where the antecedent is acting as the subject of the clause, a B connector serves to connect the noun to the verb of the relative clause. Nouns have a B+ connector. Notice that this is optional; it is also &ed with the S+, SI-, O+, and J+ connectors, meaning that one of these connectors must be used whether or not the noun takes a relative clause. Verbs have a B- connector which is orred with their S- connectors; if a verb is in a subject-type relative clause, it may not make an S connection as well.

For subject-type relative clauses, the relative pronoun *who* is mandatory. For this purpose, verbs have a Z- connector anded with their B- connector. *Who* has a Z+ connector; therefore it can fulfill this need. However, it also has a C- connector anded with its Z+ connector; this must connect back to the C+ connector on nouns. This allows the following:

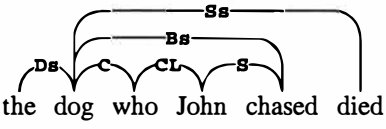


For object-type relative clauses, the same B+ connector on nouns is used. However, this time it connects to the *other* B- connector on verbs, the one which is orred with the O+ connector and which is also used for object-type *wh-* questions.

In this case, the relative pronoun *who* is optional. Notice that nouns have optional C+ and CL- connectors which are anded with their S+ connectors. These are used when the noun is the subject of an object-type relative clause. When *who* is not present, the C+ connector on the antecedent noun connects directly to the C- on the subject of the relative clause:



When *who* is present, the C+ on the antecedent connects to the C- on *who*; this forces the CL+ to connect to the CL- on the subject of the clause:



This system successfully rejects the following incorrect sentences:

*The dog chased cats died
*The dog who chase cats died
*The dog who John chased cats died
*The dog John chased cats died
*The dog who chased died

The following incorrect constructions are accepted, but can be weeded out in post-processing:

*The dog did John chase died
*The dog who John died Mary chased died

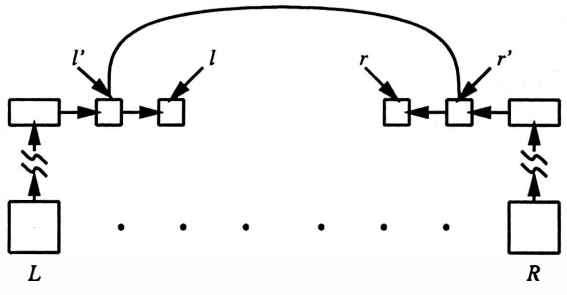
4 The algorithm

Our algorithm for parsing link grammars is based on dynamic programming. Perhaps its closest relative in the standard literature is the dynamic programming algorithm for finding an optimal triangulation of a convex polygon [2, p. 320]. It tries to build up a linkage (which we'll call a *solution* in this section) in a top down fashion: It will never add a link (to a partial solution) that is above a link already in the partial solution.

The algorithm is most easily explained by specifying a data structure for representing disjuncts. A disjunct d has pointers to two linked lists of connectors. These pointers are denoted $left[d]$ and $right[d]$. If c is a connector, then $next[c]$ will denote the next connector after c in its list. The next field of the last pointer of a list has the value NIL.

For example, suppose the disjunct $d =$((D,0) ()) (using the notation of section 2). Then $left[d]$ would point to the connector 0, $next$[left[d]] would point to the connector D, and $next$[next[$left[d]$]] would be NIL. Similarly, $right[d]$ = NIL.

To give some intuition of how the algorithm works, consider the situation after a link has been proposed between a connector l' on word L and a connector r' on word R. (The words of the sequence to be parsed are numbered from 0 to $N-1$.) For convenience, we define l and r to be $next[l']$ and $next[r']$ respectively. The situation is shown in the following diagram:



Here the square boxes above the words L and R represent a data structure node corresponding to the word. The rectangular box above each of these represents one of the (possibly many) disjuncts for the word. The small squares pointed to by the disjuncts represent connectors.

How do we go about extending the partial solution into the region strictly between L and R? (This region will be denoted $(L,\ldots,R)$.) First of all, if there are no words in this region (*i.e.* $L = R+1$) then the partial solution we've built is certainly invalid if either $l \neq$ NIL or $r \neq$ NIL. If $l = r =$ NIL then this region is ok, and we may proceed to construct the rest of the solution.

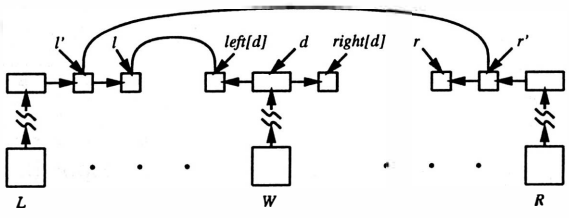
Now suppose that the region between L and R contains at least one word. In order to attach the words of this region to the rest of the sentence there must be at least one link either from L to some word in this region, or from R to some word in this region (since no word in this region can link to a word outside of the $[L,\ldots,R]$ range, and something must connect these words to the rest of the sentence).

Since the connector l' has already been used in the solution being constructed, this solution must use the rest of the connectors of the disjunct in which l' resides. The same holds for r'. The only connectors of these disjuncts that can be involved in the $(L,\ldots,R)$ region are those in the lists beginning with l and r. (The use of any other connector on these disjuncts in this region would violate the ordering requirement.) In fact, all of the connectors of these lists must be used in this region in order to have a satisfactory solution.

Suppose, for the moment, that l is not NIL. We know that this connector must link to some disjunct on some word in the region $(L,\ldots,R)$. (It can't link to R because of the exclusion rule.) The algorithm tries all possible such words and disjuncts. Suppose it finds a word W and a dis-

junct d on W such that the connector l matches *left*[d]. We can now add this link to our partial solution.

The situation is shown in the following diagram.



How do we determine if this partial solution can be extended to a full solution? We do this by solving two problems similar to the problem we started with. In particular, we ask if the solution can be extended to the word range $(L, \ldots, W)$ using the connector lists beginning with *next*[l] and *next*[left[d]]. We also ask if the solution can be extended to the word range $(W, \ldots, R)$ using the connector lists beginning with *right*[d] and r. Notice that in the latter case, the problem we are solving seems superficially different: the boundary words have not already been connected together by a link. This difference is actually of no consequence because the pair of links (L to R and L to W) play the role that a direct link from W to R would play: (1) they separate the region $(W, \ldots, R)$ from all the other words, and (2) they serve to connect the words W and R together.

We need to consider one other possibility. That is that there might be a solution with a link between words L and W *and* a link between words W and R. (This results in a solution where the word/link graph is cyclic.) The algorithm handles this possibility by also attempting to form a link between *right*[d] and r. If these two match, it does a third recursive call, solving a third problem analogous to our original problem. In this problem the word range is $(W, \ldots, R)$ and the connector lists to be satisfied begin with *next*[right[d]] and *left*[r].

A very similar analysis suffices to handle the case when l is NIL.

The algorithm described has an exponential worst-case running time as a function of N, the number of words in the sequence to be parsed. This can easily be transformed into an efficient dynamic programming algorithm by using *memoization* ([2, p. 312]).

The running time is now bounded by the number of different possible recursive calls multiplied by the time used by each call. A recursive call is completely determined by specifying the pointers l and r. (These uniquely determine L and R.) The cost of a given call is bounded by the total number of disjuncts in the sequence of words.

If we let d be the number of disjuncts and c be the number of connectors, then the running time is $O(c^2 d)$. For a fixed link grammar, $d = O(N)$ and $c = O(N)$, so the running time is $O(N^3)$.

Our technical reports describe this algorithm in more detail. They contain pseudo-code for the algorithm [14], an argument for it's correctness [14] and an elegant recurrence for the number of linkages of a sentence [11].

After the algorithm above was implemented, we were interested in seeing how well it would work on sentences taken from newspapers and other natural sources. It quickly became clear that something else was needed to make the algorithm run faster on long sentences.

5 Speeding it up

As pointed out in the introduction, in a link grammar dictionary with significant coverage of English grammar the number of disjuncts on many words gets rather large. Thus, the constant d in the analysis at the end of the last section is quite large. We devised and implemented several time-saving schemes that run in conjunction with the algorithm of the previous section.

5.1 Pruning

Our first approach is based on the following observation: In any particular sequence of words to be parsed, most of the disjuncts are irrelevant for the simple reason that they contain a connector that does not match any other connector on a word in the sequence. To be more precise, suppose that a word W has a disjunct d with a connector `C` in its right list. If no word to the right of W has a connector (pointing to the left) that matches `C`, then the disjunct d cannot be in any linkage. This disjunct can therefore be deleted without changing the set of linkages. Deleting such a disjunct is called a *pruning step*. *pruning* consists of repeating the pruning step until it can no longer be applied.

The set of disjuncts left (after pruning is complete) is independent of the order in which the steps are applied. (The pruning operation has the Church-Rosser property.) We therefore choose an ordering that can be efficiently implemented. It would be ideal if we could achieve a running time for pruning that is linear in the number of connectors. The scheme we propose satisfies no useful a-priori bound on its running time, but in practice it appears to run in linear time.

A series of sequential passes through the words is made, alternately left-to-right and right-to-left. The two types of passes are analogous, so it suffices to describe the left-to-right pass. The pass processes the words sequentially, starting with word 1. Consider the situation after words $1, \ldots, W-1$ have been processed. A set S of connectors has been computed. This is the set of connectors that exists on the right lists of the disjuncts of words $1, \ldots, W-1$ that have not been deleted. To process word W, we consider each disjunct d of W in turn. For each connector c on the left list of d, we search the set S to see if it contains a connector that matches c. If one of the connectors of d matches nothing in S, then we apply the pruning step to d (we remove d). Each right connector of each remaining disjunct of W is now incorporated into the set S. This completes the processing of word W.

The function computed by this left-to-right pass is idempotent, which is another way of saying that doing the operation twice in a row will be the same as doing it once. Therefore if (as we alternate left-to-right and right-to-left passes) a pass (after the first one) does nothing, then all further passes will do nothing. This is how the algorithm decides when to stop.

The data structure used for the set S is simply a hash table, where the hash function only uses the initial upper-case letters of the connector name. This ensures that if two connectors get hashed to different locations, then they definitely don't match.

Although we know of no non-trivial bound on the number of passes, we have never seen a case requiring more than five. Table 1 shows a typical example of the reduction in the number of disjuncts achieved by pruning.

Initial:	3	12	18	391	296	9	10	3	81	391	20	423	104	391
after L→R	3	8	11	67	160	6	10	3	81	163	8	381	25	357
after R→L	3	6	5	25	21	3	3	3	25	25	3	25	4	12
after L→R	3	6	5	25	21	3	3	3	25	21	3	21	3	8
after R→L	3	6	5	25	21	3	3	3	25	21	3	21	3	8
After P.P.:	2	6	1	13	2	2	2	2	2	6	1	4	2	1

Table 1: This table shows the number of disjuncts remaining on each word of the sentence *Now this vision is secular, but deteriorating economies will favor Islamic radicalism.* (The first number is for *the wall* which has not been described in this paper. Of course the comma also counts as a word.) The fourth pass of pruning has no effect, so pruning stops. The last row in the table shows the number of disjuncts that remain after power pruning.

5.2 The fast-match data structure

The inner loop in the algorithm described in section 4 searches for a word W and a disjunct d of this word whose first left connector matches l, or whose first right connector matches r. If there were a fast way to find all such disjuncts, significant savings might be achieved. The fast-match data structure, which is based on hashing, does precisely this. The speed-up afforded by this technique is roughly the number of different connector types, which is roughly 30 in our current dictionary.

5.3 Power pruning

Power pruning is a refinement of pruning that takes advantage of the ordering requirement of the connectors of a disjunct, the exclusion rule, and other properties of any valid linkage. It also

interacts with the fast-match data structure in a beautiful way. Unfortunately, these details are beyond the scope of this paper [4]. Table 1 shows outcome of pruning and power pruning on a typical sentence.

Each of the refinements described in this section significantly reduced the time required to do search for a linkage. The operations of pruning, power pruning, and searching for a linkage all take roughly the same amount of time.

6 Dependency and categorial grammars

6.1 Dependency formalisms

There is a large body of work based on the idea that linguistic analysis can be done by drawing links between words. These are variously called *dependency systems* [5], *dependency syntax* [10], *dependency grammar* [3, 4], or *word grammar* [6, 7].

In dependency grammar, a grammatical sentence is endowed with a *dependency structure*, which is very similar to a linkage. This structure, as defined by Melčuk [10], consists of a set of planar directed arcs among the words that form a tree. Each word (except the *root word*) has an arc out to exactly one other word, and no arc may pass over the root word. In a linkage (as opposed to a dependency structure) the links are labeled, undirected, and may form cycles, and there is no notion of a root word.

Gaifman [5] was the first to actually give a formal method of expressing a dependency grammar. He shows that his model is context-free.

Melčuk's definition of a dependency structure, and Gaifman's proof that dependency grammar is context free imply that there is a very close relationship between these systems and link grammars. This is the case.

It is easy to take a dependency grammar in Gaifman's notation and generate a link grammar that accepts the same language. In this correspondence, the linkage that results from parsing a sentence is the same as the corresponding dependency structure. This means that our algorithm for link parsing can easily be applied to dependency grammars. The number of disjuncts in the resulting link grammar is at most quadratic in the number of rules in the dependency grammar. None of the algorithms that have been described for dependency parsing [3, 15, 7] seem to bear any resemblance to ours. It is therefore plausible to conjecture that our algorithms and techniques could be very useful for directly parsing dependency grammars.

Gaifman's result shows that it is possible to represent a link grammar as a dependency grammar (they're both context-free). But this correspondence is of little use if the parsed structures that result are totally different.

One problem with constructing a dependency grammar that is in direct correspondence with a given link grammar is that a linkage in a link grammar my have cycles, whereas cycles are not allowed in dependency grammar. If we restrict ourselves to acyclic linkages, we run into another problem. This is that there is an exponential blow-up in the number of rules required to express the same grammar. This is because each disjunct of each word in the link grammar requires a separate rule in the dependency grammar.

Gaifman's model is not lexical. The method classifies the words into categories. One word can belong to many categories. Roughly speaking, for each disjunct that occurs in the dictionary, there is a category of all words that have that disjunct. The notation is therefore in a sense orthogonal to the link grammar notation.

We are not aware of any notation for dependency systems that is lexical, or that is as terse and well suited for a natural language grammar as link grammars. There has been work on creating dependency grammars for English [7, 3], but we are not aware of an implementation of a dependency grammar for any natural language that is nearly as sophisticated as ours.

6.2 Categorial grammars

Another grammatical system, known as a *categorial grammar* [1] bears some resemblance to link grammars. Below we show how to express any categorial grammar concisely as a link grammar. It appears to be more difficult to express a link grammar as a categorial grammar.

Just as in a link grammar, each word of a categorial grammar is associated with one or more

[4] Although they do appear in our technical report [14].

symbolic expressions. An expression is either an atomic symbol or a pair of expressions combined with one of two types of binary operators: / and \.

A sentence is in the language defined by the categorial grammar if, after choosing one expression associated with each word, there is a *derivation* which transforms the chosen sequence of expressions into S, a single expression consisting of a special atomic symbol. The derivation proceeds by combining two neighboring expressions into one using one of the following rules:

$$\frac{e \qquad e\backslash f}{f} \qquad\qquad \frac{f/e \qquad e}{f}$$

Here e and f are arbitrary expressions, and $f\backslash e$ and f/e are other expressions built using e and f. In both cases the two expressions being combined (the ones shown above the line) must be adjacent in the current sequence of expressions. Each combinational operation produces one expression (the one below the line), and reduces the number of expressions by one. After $n-1$ operations have been applied, a sentence of length n has be reduced to one expression.

For example, consider the following categorial grammar [9]:

```
Harry:        NP, S/(S\NP)
likes:        (S\NP)/NP
peanuts:      NP
passionately: (S\NP)\(S\NP)
```

Here is the derivation of *Harry likes peanuts passionately*.

```
  Harry      likes     peanuts    passionately
S/(S\NP)  (S\NP)/NP     NP        (S\NP)\(S\NP)
          ---------------
               S/NP
               ----------------------------
                                S\NP
 -------------------------------
                 S
```

The set of languages that can be represented by categorial grammars (as they are described here) is the set of context-free languages [1][5] This fact alone sheds no light on the way in which the formalism represents a language. To get a better understanding of the connection between categorial grammars and link grammars, the following paragraphs explain a way to construct a link grammar for a given categorial grammar. The reverse (constructing a categorial grammar from a given link grammar) seems to be more difficult, and we do not know of an elegant way to do this.

To simplify the construction, we'll use a modified definition of a link grammar called a *special link grammar*. This differs from an ordinary link grammar in two ways: the links are not allowed to form cycles, and there is a special word at the beginning of each sentence called *the wall*. The wall will not be viewed as being part of any sentence.

Let d be a categorial grammar expression. We'll show how to build an equivalent link grammar expression $E(d)$. If a word w has the set $\{d_1, d_2, \ldots, d_k\}$ of categorial expressions, then we'll give that word the following link grammar expression:

$$E(d_1)\text{or}E(d_2)\text{or}\cdots\text{or}E(d_k)$$

The function $E(\cdot)$ is defined recursively as follows:

$$\begin{aligned} E(f/e) &= f/e- \text{ or } f/e+ \text{ or } (e+ \;\&\; E(f)) \\ E(e\backslash f) &= e\backslash f- \text{ or } e\backslash f+ \text{ or } (e- \;\&\; E(f)) \\ E(A) &= A- \text{ or } A+ \end{aligned}$$

Here A stands for any atomic symbol from the categorial grammar, and A, f/e and $e\backslash f$ are connector names in the link grammar formulas.

The wall has the formula S+.

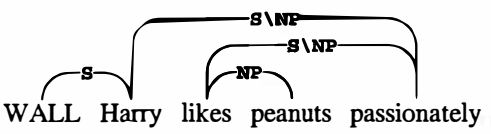
Here is the link grammar corresponding to the categorial grammar above:

```
WALL:         S+;
peanuts:      NP+ or NP-;
Harry:
  (NP- or NP+)
  or (S/<S\NP>-
     or S/<S\NP>+
     or (S\NP+ & (S+ or S-)));
likes:
  <S\NP>/NP- or <S\NP>/NP+
  or (NP+ & (S\NP- or S\NP+
             or (S- & (NP- or NP+))));
passionately:
  <S\NP>/<S\NP>- or <S\NP>/<S\NP>+
  or (S\NP- & (S\NP- or S\NP+
               or (S- & (NP- or NP+))));
```

[5]There are other variants of categorial grammars which are mildly context-sensitive[9]. Of course the construction presented here does not work for those languages.

(Here we have replaced parentheses in the categorial grammar expressions with brackets when using them inside of a link grammar expression.)

This link grammar gives the following analysis of the sentence shown above:



Notice that in this construction, both the size of the link grammar formula, and the number of disjuncts it represents are linear in the size of the original categorial grammar expressions. This suggests that a very efficient way to parse a categorial grammar would be to transform it to a link grammar, then apply the algorithms and heuristics described in this paper.

7 Remarks

Link grammars have become the basis for several other research projects. John Lafferty [11] proposes to build and automatically tune a probabilistic language model based on link grammars. The proposed model gracefully encompasses trigrams and grammatical constraints in one framework. Andrew Hunt[6] has developed a new model of the relationship of prosody and syntax based on link grammars. He has implemented the model, and in preliminary tests, the results are much better than with other models. Tom Brehony[6] has modified our parser to detect the kinds of errors that Francophones make when they write in English.

[6]Personal communication.

References

[1] Y. Bar-Hillel, Language and information; selected essays on their theory and application. Addison-Wesley, 1964.

[2] Cormen, T. H., C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press and McGraw-Hill, 1990.

[3] Fraser, N., "Parsing and dependency grammar," In: Carston, Robyn (ed.) *UCL Working Papers in Linguistics* 1, University College London, London, 1989 Pages 296–319.

[4] Fraser, N., "Prolegomena to a formal theory of dependency grammar," In: *UCL Working Papers in Linguistics* 2, University College London, London, 1990 Pages 298–319.

[5] Gaifman, H., "Dependency systems and phrase-structure systems," *Information and Control* **8**, 1965, Pages 304–337.

[6] Hudson, R., *Word Grammar*, Basil Blackwell, 1984.

[7] Hudson, R., "Towards a computer testable word grammar of English," In: Carston, Robyn (ed.) *UCL Working Papers in Linguistics* 1, University College London, London, 1989, Pages 321–339.

[8] Hudson, R., *English Word Grammar*, Basil Blackwell, 1990.

[9] Joshi, A. K., "Natural Language Processing," *Science*, Volume 253, No. 5025, (Sept. 13, 1991), Pages 1242–1249.

[10] Mel'čuk, I. A. *Dependency Syntax: Theory and Practice*, State University of New York Press 1988.

[11] Lafferty, J, D. Sleator, D. Temperley, "Grammatical Trigrams: A Probabilistic Model of Link Grammar," Proc. of the 1992 AAAI Fall Symp. on Probabilistic Approaches to Natural Language, and Technical report CMU-CS-92-181, School of Computer Science, Carnegie Mellon University, Sept 1992.

[12] Oehrle, R. T., E. Bach, and D. Wheeler, Editors *Categorial Grammars and Natural Language Structures* D. Reidel Publishing Company, 1988.

[13] Y. Schabes. "Stochastic lexicalized tree-adjoining grammars." In *Proceedings of COLING-92*, Nantes, France, July 1992.

[14] Sleator, D. D., D. Temperley, "Parsing English with a Link Grammar," Technical report CMU-CS-91-196, Carnegie Mellon University, School of Computer Science, October 1991.

[15] van Zuijlen, J., M., "Probabilistic methods in dependency grammar parsing," Proceedings of the International Workshop on Parsing Technologies, Carnegie Mellon University, 1989, Pages 142–250.

Evaluation of TTP Parser: A Preliminary Report

Tomek Strzalkowski* and Peter G. N. Scheyen‡

* Courant Institute of Mathematical Sciences, New York University
715 Broadway, rm 704, New York, NY 10003
email: tomek@cs.nyu.edu

‡ Department of Computer Science, The University of Western Ontario
London, Ontario N6A 5B7
email: scheyen@csd.uwo.ca

Abstract

TTP (Tagged Text Parser) is a fast and robust natural language parser specifically designed to process vast quantities of unrestricted text. TTP can analyze written text at the speed of approximately 0.3 sec/sentence, or 73 words per second. An important novel feature of TTP parser is that it is equipped with a skip-and-fit recovery mechanism that allows for fast closing of more difficult sub-constituents after a preset amount of time has elapsed without producing a parse. Although a complete analysis is attempted for each sentence, the parser may occasionally ignore fragments of input to resume "normal" processing after skipping a few words. These fragments are later analyzed separately and attached as incomplete constituents to the main parse tree. TTP has recently been evaluated against several leading parsers. While no formal numbers were released (a formal evaluation is planned later this year), TTP has performed surprisingly well. The main argument of this paper is that TTP can provide a substantial gain in parsing speed giving up relatively little in terms of the quality of output it produces. This property allows TTP to be used effectively in parsing large volumes of text.

1 Overview of This Paper

Recently, there has been a growing demand for fast and reliable natural language processing tools, capable of performing reasonably accurate syntactic analysis of large volumes of text within an acceptable time. A full sentential parser that produces complete analysis of input, may be considered reasonably fast if the average parsing time per sentence falls anywhere between 2 and 10 seconds. A large volume of text, perhaps a gigabyte or more, would contain as many as 7 million sentences. At the speed of say, 6 sec/sentence, this much text would require well over a year to parse. While 7 million sentences is a lot of text, this much may easily be contained in a fair-sized text database. Therefore, the parsing speed would have to be increased by at least a factor of 10 to make such a task manageable.

In this paper we describe TTP, a fast and robust natural language parser that can analyze written text and generate regularized parse structures at a speed of below 1 second per sentence. In the experiments conducted on variety of natural language texts, including technical prose, news messages, and newspaper articles, the average parsing time varied between 0.3 sec/sentence and 0.5 sec/sentence, or between 2500 and 4300 words per minute, as we tried to find an acceptable compromise between parser's speed and precision (these results were obtained on a Sun SparcStation 2). Original experiments were performed within an information retrieval system with the recall/precision statistics used to measure effectiveness of the parser.

In the second part of the paper, the linguistic

accuracy of TTP is discussed based on the partial results of a quantitative evaluation of its output using the Parseval method (Black et al, 1991). This method calculates three scores of "closeness" as it compares the bracketed parse structures returned by the parser against a pre-selected standard. These scores are: the crossings rate which indicates how many constituents in the candidate parse are incompatible with those in the standard; recall which is the percentage of candidate constituents found in the standard; and precision which specifies the percentage of standard constituents in the candidate parse. Parseval may also be used to compare the performance of different parsers. In comparison with NYU's Proteus parser, for example, which is on average two levels of magnitude slower than TTP, the crossing score was only 6 to 27% higher for TTP, with recall 13% lower, and approximately the same precision for both parsers.

In addition we discuss the relationships between allotted parse time per sentence, the average parsing time, crossings rate, and recall and precision scores.

2 Introduction to Tagged Text Parser

It has long been assumed that in order to gain speed, one may have to trade in some of the parser's accuracy. For example, we may have to settle for partial parsing that would recognize only selected grammatical structures (e.g. noun phrases; Ruge et al., 1991), or would avoid making difficult decisions (e.g. pp-attachment; Hindle, 1983). Much of the overhead and inefficiency comes from the fact that the lexical and structural ambiguity of natural language input can only be dealt with using limited context information available to the parser. Partial parsing techniques have been used with a considerable success in processing large volumes of text, for example AT&T's Fidditch (Hindle and Rooth, 1991) parsed 13 million words of Associated Press news messages, while MIT's parser (de Marcken, 1990) was used to process the 1 million word Lancaster/Oslo/Bergen (LOB) corpus. In both cases, the parsers were designed to do partial processing only, that is, they would never attempt a complete analysis of certain constructions, such as the attachment of pp-adjuncts, subordinate clauses, or coordinations. This kind of partial analysis may be sufficient in some applications because of a relatively high precision of identifying correct syntactic dependencies, for example Church and Hanks (1990) used partial parses generated by Fidditch to study word co-occurrence patterns in syntactic contexts. On the other hand, applications involving information extraction or retrieval from text will usually require more accurate parsers.

An alternative is to create a parser that would attempt to produce a complete parse, and would resort to partial or approximate analysis only under exceptional conditions such as an extragrammatical input or a severe time pressure. Encountering a construction that it couldn't handle, the parser would first try to produce an approximate analysis of the difficult fragment, and then resume normal processing for the rest of the input. The outcome is a kind of "fitted" parse, reflecting a compromise between the actual input and grammar-encoded preferences. One way to accomplish this is to adopt the following procedure: (1) close (reduce) the obstructing constituent (one which is being currently parsed), then possibly reduce a few of its parent constituents, removing corresponding productions from further consideration, until a production is reactivated for which a continuation is possible; (2) Jump over the intervening material so as to restart processing of the remainder of the sentence using the newly reactivated production.

As an example, consider the following sentence where the highlighted fragment is likely to cause problems, and may be better off ignored in the first pass:

> "The method is illustrated by the automatic construction of both recursive and iterative programs operating on natural numbers, lists, and trees, in order to construct a program satisfying certain specifications *a theorem induced by those specifications is proved,* and the desired program is extracted from the proof."

Assuming that the parser now reads the article 'a' following the string 'certain specifications', it may proceed to reduce the current NP, then SI → to"+V+NP, SI → SA, SA → NP+V+NP+SA,

until finally S → S+"and"+S is reached on the stack. Subsequently, the parser skips input to find 'and', then resumes normal processing. The key point here is that TTP decides (or is forced) to reduce incomplete constituents rather than to backtrack or otherwise select an alternative analysis. However, this is done only after the parser is thrown into the panic mode, which in case of TTP is induced by the time-out signal. In other words, while there is still time TTP will proceed in regular top-down fashion until the time-out signal is received. Afterwards, for some productions early reduction will be forced and fragments of input will be skipped if necessary. If this action does not produce a parse within a preset time (which is usually much longer than the original 'regular' parsing time), the second time-out signal is generated which forces the parser to finish even at the cost of introducing dummy constituents into the parse tree. The skipped-over fragments of input are quickly processed by a simple phrasal analyzer, and then attached to the main parse tree at the points where they were deleted.

An example parse structure returned by TTP is shown below. Note (vrbtm X) brackets which surround all un-parsed tokens in the input.

Sentence:

> Mrs. Hills said that the U.S. is still concerned about "disturbing developments in Turkey and continuing slow progress in Malaysia."

TTP Approximate Parse:

```
(sent
   (np
      (name
         mrs
         .
         hills))
   (vp
      (verb said)
      (thats
         (compl that)
         (sent
            (np
               (t_pos the)
               (name u.s.))
            (vp
               (verb
                  is
                  (adv still))
               (venpass
                  (verb concerned)))))
   ((vrbtm about)
      (vrbtm ")
      (((np
         (adj disturbing)
         (n developments)))
         (pp
         (prep in)
         (np
            (np
               (name turkey))
            and
            (np
               (a_pos_v continuing)
               (adj slow)
               (n progress))))
         (pp
         (prep in)
         (np
            (name
               malaysia
               .))))
      (vrbtm "))))
```

As may be expected, this kind of action involves a great deal of indeterminacy which, in case of natural language strings, is compounded by the high degree of lexical ambiguity. If the purpose of this skip-and-fit technique is to get the parser smoothly through even the most complex strings, the amount of additional backtracking caused by the lexical level ambiguity is certain to defeat it. Without lexical disambiguation of input, the parser's performance will deteriorate, even if the skipping is limited only to certain types of adverbial adjuncts. The most common cases of lexical ambiguity are those of a plural noun (nns) vs. a singular verb (vbz), a singular noun (nn) vs. a plural or infinitive verb (vbp,vb), and a past tense verb (vbd) vs. a past participle (vbn), as illustrated in the following example.

> "The notation used (vbn or vbd?) explicitly associates (nns or vbz?) a data structure (vb or nn?) shared

(vbn or vbd?) by concurrent processes (nns or vbz?) with operations defined (vbn or vbd?) on it."

We use a stochastic tagger to process the input text prior to parsing. The tagger, developed at BBN (see Meteer et al., 1991) is based upon a bi-gram model and it selects most likely tag for a word given co-occurrence probabilities computed from a relatively small training set. The input to TTP looks more like the following:

"The/dt notation/nn used/vbn explicitly/rb associates/vbz a/dt data/nns structure/nn shared/vbn by/in concurrent/jj processes/nns with/in operations/nns defined/vbn on/in it/pp ./."

In a 'normal' operation, TTP produces a regularized representation of each parsed sentence that reflects the sentence's logical structure. This representation may differ considerably from a standard parse tree, in that the constituents get moved around (e.g., de-passivization), and the phrases are organized recursively around their head elements. However, for the purpose of the evaluation with Parseval an 'input-bracketing' version has been created. In this version the skipped-over material is simply left unbracketed.

As the parsing proceeds, each sentence receives a new slot of time during which its parse is to be returned. The amount of time allotted to any particular sentence can be regulated to obtain an acceptable compromise between parser's speed and accuracy. In our experiments we found that 0.5 sec/sentence time slot was appropriate for the Wall Street Journal articles (the average length of the sentence in our WSJ collection is 17 words). We must note here that giving the parser more time per sentence doesn't always mean that a better (more accurate) parse will be obtained. For complex or extra-grammatical structures we are likely to be better off if we do not allow the parser wander around for too long: the most likely interpretation of an unexpected input is probably the one generated early (the grammar rule ordering enforces some preferences). In fact, our experiments indicate that as the 'normal' parsing time is extended, the accuracy of the produced parse increases at an ever slowering pace, peaking for a certain value, then declining slightly to eventually stabilize at a constant level. This final level-off indicates, we believe, an inherent limit in the coverage of the underlying grammar.

3 The TTP Time-out Mechanism

The time-out mechanism is implemented using a straightforward parameter passing and is limited to only a subset of nonterminals used by the grammar. Suppose that X is such a nonterminal, and that it appears on the right-hand side of a production S → X Y Z. The set of "starters" is computed for Y, which consists of the word tags that can occur as the left-most constituent of Y. This set is passed as a parameter while the parser attempts to recognize X in the input. If X is recognized successfully within a preset time, then the parser proceeds to parse a Y, and nothing else happens. On the other hand, if the parser cannot determine whether there is an X in the input or not, that is, it neither succeeds nor fails in parsing X before being timed out, the unfinished X constituent is closed (reduced) with a partial parse, and the parser is restarted at the closest element from the starters set for Y that can be found in the remainder of the input. If Y rewrites to an empty string, the starters for Z to the right of Y are added to the starters for Y and both sets are passed as a parameter to X. As an example consider the following clauses in the TTP parser:

```
sentence(P) :-
 assertion([],P).
assertion(SR,P) :-
 clause(SR,P1),
 s_coord(SR,P1,P).
clause(SR,P) :-
 sa([pdt,dt,cd,pp,ppS,jj,jjr,jjs,nn,nns,np,nps],P2),
 subject([vbd,vbz,vbp],P1),
 verbphrase(SR,P1,P2,P).
thats(SR,P) :-
 that,
 assertion(SR,P).
```

In the above code, P, P1, and P2 represent partial parse structures, while SR is a set of starter word tags where the parsing will resume should the present nonterminal be timed-out. First arguments to 'assertion', 'sa', and 'sub-

ject' are also sets of starter tags. In the 'clause' production above, a (finite) clause rewrites into a left sentence adjunct ('sa'), a 'subject', and a 'verbphrase'. If 'sa' is aborted before its evaluation is complete, the parser will jump over some elements of the unparsed portion of the input looking for a word that could begin a subject phrase: a pre-determiner (pdt), a determiner (dt), a count word (cd), a pronoun (pp,ppS), an adjective (jj, jjr, jjs), a noun (nn, nns), or a proper name (np, nps). Likewise, when 'subject' is timed out, the parser will restart with 'verbphrase' at either vbz, vbd or vbp (finite forms of a verb). Note that if 'verbphrase' is timed out both 'verbphrase' and 'clause' will be closed, and the parser will restart at an element of set SR passed down to 'clause' from assertion. Note also that in the top-level production for a sentence the starter set for 'assertion' is initialized to be empty: if the failure occurs at this level, no continuation is possible.

The forced reduction and skip-over are carried out through special productions that are activated only after a preset amount of time has elapsed since the start of parsing. For example, 'subject' is defined as follows:

```
subject(SR,PG) :-
    timed_out,!,
    skip(SR),
    store(PG).
subject(SR,P) :-
    noun_phrase(SR,P).
```

When a non-terminal is timed out and the parser jumps over a non-zero length fragment of input, it is assumed that the skipped part was some sub-constituent of the reduced non-terminal (e.g., subject). Accordingly, a place holder (PG) is left in the parse structure under the node dominated by this non-terminal. This placeholder will be later filled by some material recovered from the skipped-over fragment which is put aside by store(PG). In the bracketing-only version of TTP unparsed fragments are placed verbatim within the outmost bracket of the timed-out constituent, e.g.,

```
(S
    (NP we)
    (VP
        (V receive)
        (NP more pro letters
                (VRBTM than) (VRBTM con))
    ...
    ))
```

Note that (VRBTM *) brackets are invisible to the evaluation program, as will be explained in the next section.

There are a few caveats in the skip-and-fit parsing strategy just outlined which warrant further explanation. In particular, the following problems must be resolved to assure parser's effectiveness:

(a) how to select starter tags for non-terminals, and

(b) how to select non-terminals at which input skipping can occur.

Obviously some tags are more likely to occur at the left-most position of a constituent than others. Theoretically, a subject phrase can start with a word tagged with any element from the following list (still not a complete list): pdt (predeterminer), dt (determiner), cd (numerical), jj, jjr, jjs (adjective), pp, ppS (pronoun), nn, nns (noun), np, nps (name), vbg, vbn (participle), rb (adverb), in (preposition). In practice, however, we may select only a subset of these, as shown in the 'clause' production above. Although we now risk missing the left boundary of the subject phrase in some sentences, while skipping an adjunct to their left, most cases are still covered and the chances of making a serious misinterpretation of input are significantly lower.

On the other hand certain adjunct phrases may be of little interest, possibly because of their typically low information contents, and we may choose to ignore them altogether. Thus in the 'ntovo' object string production below (where 'to' is the tag of word 'to'):

```
ntovo(SR,[P—PSA]) :-
    subject([to],P1),
    sa([to],PSA),
    tovo(SR,P1,P).
```

'sa' (sentential adjunct) material is skipped entirely if 'subject' is timed out. We must note here that this scheme will behave as expected only if there is no other 'to' tag between the skip point and the beginning of 'tovo' phrase, e.g.,

"Financial planners often urge
investors to diversify ... "
└___ NTOVO ___┘
N(P) + TO + V(P)

When this is not the case, skipping may have to be redone. As an example, consider the following fragment:

"... urge
those flying to New York to take ... "
└_____ NTOVO _____________┘
└_____ NP _____________┘ to └ VP ┘

If 'subject' of 'ntovo' is timed-out, the parser will first jump to "to (New York)", and only after failing to find a verb ("New") will redo 'skip' in order to take a longer leap to the next 'to'. This example shows that a great deal of indeterminacy still remains even in the tagged text, and that the final selection of skip points and starter tags may require some training.

A related problem is the selection of nonterminals that can be allowed to time out. This is normally restricted to various less-than-essential adjunct-type constituents, including adverbials, some prepositional phrases, relative clauses, etc. Major sentential constituents such as subject or verbphrase should not be timed (though their sub-constituents can), or we risk to generate very uninteresting parses. Note that a timed-out phrase is not lost, but its links with the main parse structure (e.g., traces in relative clauses) may be severed, though not necessarily beyond repair. Another important restriction is to avoid introduction of spurious dummy phrases, for example, in order to force an object on a transitive verb. The time-out points must be placed in such a way that while the above principles are observed, the parser is guaranteed a swift exit when in the skip-and-fit mode. In other words, we do not want the parser to get trapped in inadvertently created dead ends, hopelessly trying to fit the parse.

As an additional safety valve, a second time-out signal can be issued to catch any processes still operating beyond a reasonable time after the first time-out. In this case, a relaxed skipping protocol is adopted with skips to only major constituents, or outright to the end of the input. Dummy constituents may be introduced if necessary to close a parse fast. This, however, happens rarely if the parser is designed carefully. While parsing 4 million sentences (85 million words) of Wall Street Journal articles, the second time-out was invoked less than 10 times.

4 Parser Evaluation With Parseval

Parseval is a quantitative method of parser evaluation which compares constituent bracketings in the parse's output with a set of 'standard' bracketings. A parse is understood as a system of labeled brackets imposed upon the input sentence, with no changes in either word order or form permitted. Using three separate scores of 'crossings', recall and precision assigned to each parse (and explained in more detail below) the measure determines parser's accuracy indicating how close it is to the standard. For the purpose of this evaluation Penn Treebank bracketings have been adopted as standard.

In the rest of this section we demonstrate how Parseval typically processes a sentence. The example used here is sentence 337 from Brown Corpus, one of the set of 50 sentences used in initial evaluation. In this example, the sentence has been processed with TTP time-out set at 700 msecs.

Sentence 337

"Mr. Nixon, for his part, would
oppose intervention in Cuba without
specific provocation."

TTP PARSE (lispified)

```
(S
   (NP (NAME mr. nixon))
   (VP
      ,
      (PP
         (PREP for)
         (NP (T_POS (POSS his)) (N part)))
      ,
      (VERB would)
      (VO
```

```
        (VERB oppose)
        (NP (N intervention)))
    (PP
        (PREP in)
        (NP (N cuba)))
    (PP
        (PREP without)
        (NP (ADJ specific) (N provocation))))
.)
```

The first two steps that the evaluator takes is to delete certain kinds of lesser constituents. This is done because of a great variety of treatment of these items across different parsers, and their relatively minor role in deciding correctness of a parse. The first phase deletes the following types of token strings from the parse:

1. Auxiliaries – "might have been understood" → "understood"
2. "Not" – "should not have come" → "come"
3. Pre-infinitival "to" – "not to answer" → "answer"
4. Null categories – (NP ()) → (NP)
5. Possessive endings – "Lori's mother" → "Lori mother"
6. Word-external punctuation (quotes, commas, periods, dashes, etc.)

The revised parse of sentence 337 is shown below.

```
(S
   (NP (NAME mr nixon))
   (VP
      (PP
         (PREP for)
         (NP (T_POS (POSS his)) (N part)))
      (VERB )
      (VO
         (VERB oppose)
         (NP (N intervention)))
      (PP
         (PREP in)
         (NP (N cuba)))
      (PP
         (PREP without)
         (NP (ADJ specific) (N provocation))))
   )
```

Subsequently, certain parenthesis pairs are recursively deleted, namely those that enclose either a single constituent or word, or an empty string. After this phase, sentence 337 looks like the following:

```
(S
   (NP mr nixon)
   (VP
      (PP for
         (NP his part))
      (VO oppose intervention)
      (PP in cuba)
      (PP without
         (NP specific provocation))))
```

Now the parse can be compared against the standard, which is shown below:

```
(S
   (NP mr nixon)
   (PP for
      (NP his part))
   (VP oppose
      (NP intervention
         (PP in cuba)
         (PP without
            (NP specific provocation)))))
```

We may note that there are several differences between the two structures, the most apparent is that various PP phrases are made part of VP in one of them while not in the other. In addition, TTP's VO constituent creates a "crossing" fault with respect to the standard, as it encompasses neither a subset nor a superset of any standard bracketing.

... (VO oppose intervention) ...

... (VP oppose (NP intervention ...

Other measures of closeness between these two structures are recall and precision, defined as follows:

$$recall = \frac{\#\ standard\ constituents\ in\ candidate}{\#\ constituents\ in\ standard}$$

$$prec = \frac{\#\ candidate\ constituents\ in\ standard}{\#\ constituents\ in\ candidate}$$

In the current example, both the candidate and the standard parses have 9 constituents. Seven of these constituents are common to one another, and there is one crossing fault. Therefore this TTP parse is evaluated as follows:

crossings	=	1
recall	=	77.78%
precision	=	77.78%

5 Evaluation of TTP

Two sets of evaluations were performed with TTP. In the first set, 50 sentences from Brown Corpus were used. In the series of runs with time-out value ranging from 100 to 2500 msecs, TTP scores varied from average crossing rate of 1.38 (for 100 msec time-out) to 0.82 (at 1700 msec); recall from the low of 59.37 (at 1800 msec!) to 62.02 (at 500 msec); and precision from 70.52 (at 100 msec) to 77.06 (at 1400 msec). The mean scores were: crossing rate 0.92, recall 60.57% and precision 75.69%. These scores reflect both the quality of the parser as well as the differences between grammar systems used in TTP and in preparing the standard parses. For example, average recall scores for hand parsed Brown sentences varied from 79% (for LSP grammar on which TTP is based; Sager, 1981) to 97%, with the average of 94%. We also plotted average parsing time per sentence for various time-out values. This is summarized in the table below (timing values are in milliseconds). These results were obtained on Sun SparcStationELC.

T/O	C	R	P	TIME
100	1.38	61.54	70.52	160
200	1.24	60.10	72.05	200
500	1.00	62.02	75.22	300
600	0.88	62.02	76.33	316
1000	0.90	61.78	76.49	380
1500	0.86	59.86	75.91	420
2000	0.86	59.37	76.00	460
2500	0.82	60.34	76.76	490
Mean	0.92	60.57	75.69	

The second set of tests involved 100 sentences form Wall Street Journal. Since WSJ sentences were usually longer and more complex than the 50 Brown sentences, we used time-outs of 250 msecs and more. The table below summarizes average crossings, recall and precision scores for TTP. Note that the performance peaks at time-out 500 and 750 msecs. These results are for a Sun SparcStation2.

T/O	C		R	P	TIME
	TOT	PER			
250	71	2.91	55.08	61.50	305
500	70	2.60	55.22	63.16	438
600	72	2.68	54.20	62.51	477
750	69	2.57	55.22	63.85	540
1500	68	2.60	54.57	64.01	797
30000	59	2.17	51.79	66.26	2930

The above statistics can be compared with those obtained parsing the same set of 100 sentences with a 'regular' parser: NYU's Proteus Parser (Grishman, 1990). Both parsers are based on the same grammar system (although Proteus grammar provides a much better coverage of English).

Proteus Statistics

C		R	P	TIME
TOT	PER			
56	2.34	63.74	62.87	24000

TTP Best Statistics

T/O	C		R	P	T
	TOT	PER			
500	70	2.60	55.22	63.16	438
% change					
	+25%	+11	-13	+0.4	
750	69	2.57	55.22	63.85	540
% change					
	+23%	+9.8	-13	+1.5	
30000	59	2.17	51.79	66.26	2930
% change					
	+5%	-7	-19	+5	

One should note that the per-sentence crossing ratio and recall score indicate how good the parser is at discovering the correct bracketings (precision is less useful as it already includes crossing errors). Clearly, both the crossing ratio and precision improves as we let the parser take more time to complete its job. On the other hand, the recall, after an initial increase, declines somewhat for larger values of time-out. This, we believe,

points to the limitations of the underlying grammar used in these tests: initial correct hypotheses (enforced by preferences within the parser) are replaced by less likely ones when the parser is forced to backtrack.

6 Conclusions

At present TTP is a part of a natural language information retrieval system. Along with a stochastic part-of-speech tagger, morpho-lexical stemmer and phrase extractor, it constitutes the linguistic pre-processor built on top of the statistical information retrieval system PRISE, developed at NIST. During the database creation process TTP is used to parse documents so that appropriate indexing phrases can be identified with a high degree of accuracy. Both phrases and single-word terms are selected, along with their immediate syntactic context which is used to generate semantic word associations and create a domain-specific level-1 thesaurus. For TREC-1 conference concluded last November, the total of 500 MBytes of Wall Street Journal articles have been parsed. This is approximately 4 million sentences, and it took about 2 workstation-weeks to process. While the quality of parsing was less than perfect, it was nonetheless quite good. In various experiments with the final database we noted an increase of retrieval precision over the purely statistical base system that ranged from 6% (notable) to more than 13% (significant). Therefore, at least in applications such as document retrieval or information extraction, TTP-level parsing appears entirely sufficient, while its high speed and robustness makes an otherwise impractical task of linguistic text processing, quite manageable, and moreover, at par with other statistical parts of the system.

Further development of TTP will continue, especially expanding its base grammar to bring the coverage closer to Sager's LSP or Grishman's Proteus. We are also investigating ways of automated generation of skipping parsers like TTP out of any full grammar parser, a process we call 'ttpization'. TTP has been made available for research purposes to several sites outside NYU, where it is used in variety of applications ranging from information extraction from text to optical readers.

Acknowledgements

This paper is based upon work supported by the Advanced Research Project Agency under Contract N00014-90-J-1851 from the Office of Naval Research, Contract N00600-88-D-3717 from PRC Inc., the National Science Foundation under Grant IRI-89-02304, and by the Canadian Institute for Robotics and Intelligent Systems (IRIS).

References

Church, Kenneth Ward and Patrick Hanks. 1990. "Word association norms, mutual information, and lexicography." Computational Linguistics, 16(1), MIT Press, pp. 22–29.

De Marcken, Carl G. 1990. "Parsing the LOB corpus." Proceedings of the 28th Meeting of the ACL, Pittsburgh, PA. pp. 243–251.

Grishman, Ralph. 1990. "Proteus Parser Reference Manual." Proteus Project Memorandum #4-C, Courant Institute of Mathematical Sciences, New4-C, Courant Institute of Mathematical Sciences, New4-C, Courant Institute of Mathematical Sciences, New York University.

Harrison, P., S. Abney, E. Black, D. Flickinger, C. Gdaniec, R. Grishman, D. Hindle, R. Ingria, M. Marcus, B. Santorini, T. Strzalkowski. 1991. "Evaluating Syntax Performance of Parser/Grammars of English." Natual Language Processing Systems Evaluation Workshop, Berkeley, CA. pp. 71–78.

Hindle, Donald. 1983. "User manual of Fidditch, a deterministic parser." Naval Research Laboratory Technical Memorandum 7590-142.

Hindle, Donald and Mats Rooth. 1991. "Structural Ambiguity and Lexical Relations." Proceedings of the 29th Meeting of the ACL, Berkeley, CA. pp. 229–236.

Meteer, Marie, Richard Schwartz, and Ralph Weischedel. 1991. "Studies in Part of Speech Labeling." Proceedings of the 4th DARPA Speech and Natural Language Workshop, Morgan-Kaufman, San Mateo, CA. pp. 331–336.

Ruge, Gerda, Christoph Schwarz, Amy J. Warner. 1991. "Effectiveness and Efficiency in Natural Language Processing for Large Amounts of Text." Journal of the ASIS, 42(6), pp. 450-456.

Sager, Naomi. 1981. *Natural Language Information Processing.* Addison-Wesley.

Strzalkowski, Tomek and Barbara Vauthey. 1992. "Information Retrieval Using Robust Natural Language Processing." Proceedings of the 30th ACL Meeting, Newark, DE, June-July. pp. 104–111.

Strzalkowski, Tomek. 1992. "TTP: A Fast and Robust Parser for Natural Language." Proceedings of COLING-92, Nantes, France, July 23–28.

Strzalkowski, Tomek. 1992. "Natural Language Processing in Large-Scale Text Retrieval Tasks." First Text Retrieval Conference (TREC-1), Rockville, Md, November 4–6.

Appendix: Example Parses Produced by TTP

All sentences in the examples below are from Wall Street Journal sample. Parses obtained at 750 msecs time-out on Sun SparcStation2.

Sentence:

Mr. McGovern, 63, had been under intense pressure from the board to boost Campbell's mediocre performance to the level of other food companies.

TTP PARSE:

```
((sent
  (np
    (name
      mr
      .
      mcgovern)
    (rn
      (((punct ,))
        (appos
          (np
            (count 63)))
        ,)))
  (vp
    (verb had)
    (veno
      (verb been)
      ((pp
        (prep under)
        (np
          (adj intense)
          (n pressure))))
      ((pp
        (prep from)
        (np
          (t_pos the)
          (n board))))
      (tovo
        (prep to)
        (tvp
          (verb boost)
          (np
            (t_pos
              (poss
                ((name campbell))
                's))
            (adj mediocre)
            (n performance))
          ((pp
            (prep to)
            (np
              (t_pos the)
              (n level)
              (rn
                (pp
                  (prep of)
                  (np
                    (adj other)
                    (n_pos
                      (np
                        (n food)))
                    (n companies))))))))))
  .)
```

CROSSINGS: 1
RECALL : 86.67%
PRECISION: 81.25%

Sentence:

Any money in excess of $40 million collected from the fees in fiscal 1990 would go to the Treasury at large.

TTP PARSE:

```
((sent
  (np
    (t_pos any)
    (n money))
  (vp
    ((pp
      (prep in)
      (np
        (n excess)
        (rn
          (pp
            (prep of)
            (np
              (n
                $
                40
                million)
              (rn
                (rn_wh
                  (venpass
                    (verb collected)
                    ((pp
                      (prep from)
                      (np
                        (t_pos the)
                        (n fees))))
                    ((pp
                      (prep in)
                      (np
                        (count
                          fiscal
                          1990)))))))))))))
    (verb would)
    (vo
      (verb go)
      (pp
        (prep to)
        (np
          (t_pos the)
          (n treasury))))
    ((pp
      (prep at)
      (np
        (n large))))))
.)
```

CROSSINGS: 7
RECALL : 50.00%
PRECISION: 47.06%

Sentence:

RJR Nabisco Inc. and American Brands Inc. say they have no plans to follow Philip Morris's lead.

TTP PARSE:

```
((sent
  (np
    (np
      (name
        rjr
        nabisco
        inc
        .))
      and
      (np
        (name
          american
          brands
          inc
          .)))
  (vp
    (verb say)
    (thats
      (compl ())
(sent
  (np
    (n they))
  (vp
    (verb have)
    (np
      (t_pos no)
      (n plans)
      (rn
        (rn_wh
          (tovo
            (prep to)
            (tvp
              (verb follow)
              (np
                (t_pos
                  (poss
                    ((name
                      philip
                      morris))
                    's))
                (n lead))))))))))))
.)
```

CROSSINGS: 0
RECALL : 100%
PRECISION: 100%

Sentence:

> The Health Insurance Association of America, an insurers' trade group, acknowledges that stiff competition among its members to insure businesses likely to be good risks during the first year of coverage has aggravated the problem in the small-business market.

TTP PARSE:

```
((sent
 (np
  (t_pos the)
  (n_pos
   (np
    (n health))
   (n_pos
    (np
     (n insurance))))
  (n association)
  (rn
   (pp
    (prep of)
    (np
     (name america)))
   (rn
    (((punct ,))
     (appos
      (np
       (t_pos an)
       (n_pos
        (poss
         (n insurers)
         ')
        (n_pos
         (np
          (n trade))))
       (n group)))
     ,))))
 (vp
  (verb acknowledges)
  (np
   (t_pos that)
   (adj stiff)
   (n competition))
  ((pp
   (prep among)
   (np
    (t_pos
     (poss its))
    (n members)
    (rn
     (rn_wh
      (tovo
       (prep to)
       (tvp
        (verb insure)
        (np
         (n businesses)
         (rn
          (rn_wh
           ((verb likely)))
          (rn
           (rn_wh
            (tovo
             (prep to)
             (tvp
                (verb be)
                (objbe
                   ((np
                      (adj good)
                      (n risks))))))))))))))))
((pp
 (prep during)
 (np
  (t_pos the)
  (adj first)
  (n year)
  (rn
   (pp
    (prep of)
    (np
     (n coverage)
     (((vrbtm has))
      ((wh_rel
       (venpass
        (verb aggravated))))
      (((np
         (t_pos the)
         (n problem)))
       ((pp
         (prep in)
         (np
          (t_pos the)
          (n_pos
           (np
            (n small-business)))
```

```
            (n market)))))))))))))))
 .)
```

CROSSINGS: 13
RECALL : 38.46%
PRECISION: 37.04%

Sentence:

All three major creditors – the IRS, Minpeco and Manufacturers Hanover – voted against and effectively doomed a reorganization plan proposed by Mr. Hunt.

TTP PARSE:

```
(((np
    (t_pos all)
    (count three)
    (adj major)
    (n creditors)))
  (vrbtm –)
  ((np
      (t_pos the)
      (name irs)))
  (vrbtm ,)
  ((np
      (name minpeco)))
  (vrbtm and)
  ((np
      (name
        manufacturers
        hanover)))
  (vrbtm –)
  (vrbtm voted)
  (vrbtm against)
  (vrbtm and)
  (vrbtm effectively)
  (wh_rel
      (venpass
        (verb doomed)))
  ((np
      (t_pos a)
      (n_pos
        (np
          (n reorganization)))
      (n plan)
      (rn
        (rn_wh
          (venpass
            (verb proposed)
            (pp
              (prep by)
              (np
                (name
                  mr
                  .
                  hunt))))))))
 .)
```

CROSSINGS: 0
RECALL : 57.14%
PRECISION: 100.00%

Frequency Estimation of Verb Subcategorization Frames Based on Syntactic and Multidimensional Statistical Analysis

Akira Ushioda, David A. Evans, Ted Gibson, Alex Waibel

Computational Linguistics Program
Carnegie Mellon University, Pittsburgh, PA 15213-3890
email: aushioda@caesar.lcl.cmu.edu

Abstract

We describe a mechanism for automatically estimating frequencies of verb subcategorization frames in a large corpus. A tagged corpus is first partially parsed to identify noun phrases and then a regular grammar is used to estimate the appropriate subcategorization frame for each verb token in the corpus. In an experiment involving the identification of six fixed subcategorization frames, our current system showed more than 80% accuracy. In addition, a new statistical method enables the system to learn patterns of errors based on a set of training samples and substantially improves the accuracy of the frequency estimation.

1 Introduction

When we construct a grammar, there is always a trade-off between the coverage of the grammar and the ambiguity of the grammar. If we hope to develop an efficient high-coverage parser for unrestricted texts, we must have some means of dealing with the combinatorial explosion of syntactic ambiguities. While a general probabilistic optimization technique such as the Inside-Outside algorithm (Baker 1979, Lauri and Young 1990, Jelinek *et al.* 1990, Carroll and Charniak 1992) can be used to reduce ambiguity by providing estimates on the applicability of the context-free rules in a grammar (for example), the algorithm does not take advantage of lexical information, including such information as verb subcategorization frame preferences. Discovering or acquiring lexically-sensitive linguistic structures from large corpora may offer an essential complementary approach.

Verb subcategorization (verb-subcat) frames represent one of the most important elements of grammatical/lexical knowledge for efficient and reliable parsing. At this stage in the computational-linguistic exploration of corpora, dictionaries are still probably more reliable than automatic acquisition systems as a source of subcategorization (subcat) frames for verbs. The Oxford Advanced Learners Dictionary (OALD) (Hornby 1989), for example, uses 32 verb patterns to describe a usage of each verb for each meaning of the verb. However, dictionaries do not provide quantitative information such as how often each verb is used with each of the possible subcat frames. Since dictionaries are repositories, primarily, of what is possible, not what is most likely, they tend to contain information about rare usage (de Marken 1992). But without information about the frequencies of the subcat frames we find in dictionaries, we face the prospect of having to treat each frame as equiprobable in parsing. This can lead to serious inefficiency. We also know that the frequency of subcat frames can vary by domain; frames that are very rare in one domain can be quite common in another. If we could automatically determine the frequencies of subcat frames for domains, we would be able to tailor parsing with domain-specific heuristics. Indeed, it would be desirable to have a subcat dictionary for each possible domain.

parsing with domain-specific heuristics. Indeed, it would be desirable to have a subcat dictionary for each possible domain.

This paper describes a mechanism for automatically acquiring subcat frames and their frequencies based on a tagged corpus. The method utilizes a tagged corpus because (i) we don't have to deal with a lexical ambiguity (ii) tagged corpora in various domains are becoming readily available and (iii) simple and robust tagging techniques using such corpora recently have been developed (Church 1988, Brill 1992).

Brent reports a method for automatically acquiring subcat frames but without frequency measurements (Brent and Berwick 1991, Brent 1991). His approach is to count occurrences of those unambiguous verb phrases that contain no noun phrases other than pronouns or proper nouns. By thus restricting the "features" that trigger identification of a verb phrase, he avoids possible errors due to syntactic ambiguity. Although the rate of false positives is very low in his system, his syntactic features are so selective that most verb tokens fail to satisfy them. (For example, verbs that occurred fewer than 20 times in the corpus tend to have no co-occurrences with the features.) Therefore his approach is not useful in determining verb-subcat frame frequencies.

To measure frequencies, we need, ideally, to identify a subcat frame for each verb token in the corpus. This, in turn, requires a full parse of the corpus. Since manually parsed corpora are rare and typically small, and since automatically parsed corpora contain many errors (given current parsing technologies), an alternative source of useful syntactic structure is needed. We have elected to use partially parsed sentences automatically derived from a lexically-tagged corpus. The partial parse contains information about minimal noun phrases (without PP attachment or clausal complements). While such derived information about syntactic structure is less accurate and complete than that available in certified, hand-parsed corpora, the approach promises to generalize and to yield large sample sizes. In particular, we can use partially parsed corpora to measure verb-subcat frame frequencies.

b:	sentence initial maker	e:	sentence final maker
k:	target verb	t:	"to"
i:	pronoun	m:	modal
n:	noun phrase	w:	relative pronoun
v:	finite verb	a:	adverb
u:	participial verb	x:	punctuation
d:	base form verb	c:	complementizer "that"
p:	preposition	s:	the rest

Table 1: List of Symbols/Categories

2 Method

The procedure to find verb-subcat frequencies, automatically, is as follows.

1) Make a list of verbs out of the tagged corpus.

2) For each verb on the list (the "target verb"),

(2.1) Tokenize each sentence containing the target verb in the following way:
All the noun phrases except pronouns are tokenized as "n" by a noun phrase parser and all the rest of the words are also tokenized following the schmema in Table 1. For example, the sentence "The corresponding mental-state verbs do not follow [target verb] these rules in a straightforward way" is transformed to a sequence of tokens "bnvaknpne".

(2.2) Apply a set of subcat extraction rules to the tokenized sentences. These rules are written as regular expressions and

they are obtained through the examination of occurrences of a small sample of verbs in a training text.

Note that in the actual implementation of the procedure, all of the redundant operations are eliminated. Our NP parser also uses a finite-state grammar. It is designed especially to support identification of verb-subcat frames. One of its special features is that it detects time-adjuncts such as "yesterday", "two months ago", or "the following day", and eliminates them in the tokenization process. For example, the sentence "He told the reporters the following day that..." is tokenized to "bivnc..." instead of "bivnnc...".

3 Experiment on Wall Street Journal Corpus

We used the above method in experiments involving a tagged corpus of Wall Street Journal (WSJ) articles, provided by the Penn Treebank project. Our experiment was limited in two senses. First, we treated all prepositional phrases as adjuncts. (It is generally difficult to distinguish complement and adjunct PPs.) Second, we measured the frequencies of only six fixed subcat frames for verbs in non-participle form. (This does not represent an essential shortcoming in the method; we only need to have additional subcat frame extraction rules to accommodate participles.)

	Frame	Rule
1.	NP+NP	`k(i|n)n`
2.	NP+CL	`k(i|n(pn)*)c`
		`k(i|n)(i|n)a*(m|v)`
3.	NP+INF	`k(i|n(pn)*)ta*d`
4.	CL	`kc`
		`k(i|n)a*(m|v)`
5.	NP	`k(i|n)/[^mvd]`
		`#pw(i|n(pn)*)a*m?a*k/[^t]`
6.	INF	`kta*d`

Notes:

NP: noun phrase
CL: that-clause with and without the complementizer "that"
INF: "to" + infinitive

`x*` matches a sequence of any number of x's including zero
`x?` is either x or empty
`(x|y)` matches either x or y
`[^xyz]` matches any token except x, y, and z
`#x`(sequence) matches (sequence) that is not directly preceded by x
`x/y` matches x if x is immediately followed by y

Sample Sentences:

Frame 1. "...gives current management enough time to work on..."

Frame 2. "...tell the people in the hall that..."; "...told him the man would..."

Frame 3. "...expected the impact from the restructuring to make..."

Frame 4. "...think that..."; "...thought the company eventually responded..."

Frame 5. "...saw the man..."; "...which the president of the company wanted..."
but not
"...saw him swim..."; "...(hotel) in which he stayed..."; "...(gift) which he expected to get..."

Frame 6. "...expects to gain..."

Table 2: Set of Subcategorization Frame Extraction Rules

We extracted two sets of tagged sentences from the WSJ corpus, each representing 3-MBytes and approximately 300,000 words of text. One set was used as a training corpus, the other as a test corpus. Table 2 gives the list of verb-subcat frame extraction rules obtained (via examination) for four verbs "expect", "reflect", "tell", and "give", as they occurred in the training corpus. Sample sentences that can be captured by each set of rules are attached to the list. Table 3 shows the result of the hand comparison of the automatically identified verb-subcat frames for "give" and "expect" in the test corpus. The tabular columns give actual frequencies for each verb-subcat frame based on manual review and the tabular rows give the frequencies as determined automatically by the system. The count of each cell ($[i, j]$) gives the number of occurrences of the verb that are assigned the i-th subcat frame by the system and assigned the j-th frame by manual review. The frame/column labeled "REST" represents all other subcat frames, encompassing such subcat frames as those involving wh-clauses, verb-particle combinations (such as "give up"), and no complements.

Outpput of System
"Give"

Real Occurrences

	NP+NP	NP+CL	NP+INF	NP	CL	INF	REST	Total
NP+NP	52	0	0	0	0	0	0	52
NP+CL	1	0	0	0	0	0	0	1
NP+INF	2	0	0	0	0	0	0	2
NP	13	0	0	27	0	0	0	40
CL	0	0	0	0	0	0	0	0
INF	0	0	0	0	0	0	0	0
REST	1	0	0	4	0	0	9	14
Total	69	0	0	31	0	0	9	109

Output of System
"Expect"

Real Occurrences

	NP+NP	NP+CL	NP+INF	NP	CL	INF	REST	Total
NP+NP	0	0	0	0	0	0	0	0
NP+CL	0	0	0	0	0	0	0	0
NP+INF	0	0	55	1	0	0	0	56
NP	0	0	4	28	0	0	0	32
CL	0	0	0	0	8	0	0	8
INF	0	0	0	0	0	40	0	40
REST	0	0	1	6	0	0	7	14
Total	0	0	60	35	8	40	7	150

Table 3: Subcategorization Frame Frequencies

Despite the simplicity of the rules, the frequencies for subcat frames determined under automatic processing are very close to the real distributions. Most of the errors are attributable to errors in the noun phrase parser. For example, 10 out of the 13 errors in the [NP,NP+NP] cell under "give" are due to noun phrase parsing errors such as the misidentification of a N–N sequence (e.g., *"give [NP government officials rights] against the press" vs. "give [NP government officials] [NP rights] against the press").

To measure the total accuracy of the system, we randomly chose 33 verbs from the 300 most frequent verbs in the test corpus (given in Table 4), automatically estimated the subcat frames for each occurrence of these verbs in the test corpus, and compared the results to manually determined subcat frames.

The overall results are quite promising. The total number of occurrences of the 33 verbs in the test corpus (excluding participle forms) is 2,242. Of these, 1,933 were assigned correct subcat frames by the system. (The 'correct'-assignment counts always appear in the diagonal cells in a comparison table such as in Table 3.) This indicates an overall accuracy for the method of 86%.

If we exclude the subcat frame "REST" from our statistics, the total number of occurrences of the 33 verbs in one of the six subcat frames is 1,565. Of these, 1,311 were assigned correct subcat frames by the system. This represents 83% accuracy.

For 30 of the 33 verbs, both the first and the second (if any) most frequent subcat frames as determined by the system were correct. For all of the verbs except one ("need"), the most frequent frame was correct.

Figure 1 is a histogram showing the number of verbs within each error-rate zone. In com-

acquire	end	like	spend
build	expand	need	total
close	fail	produce	try
comment	file	prove	use
consider	follow	reach	want
continue	get	receive	work
design	help	reduce	
develop	hold	see	
elect	let	sign	

Table 4: Verbs Tested

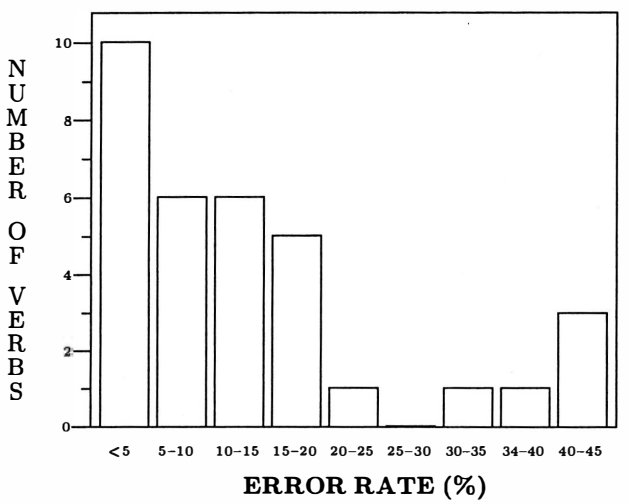


Figure 1: Distribution of Errors

puting the error rate, we divide the total 'off-diagonal'-cell counts, excluding the counts in the "REST" column, by the total cell counts, again excluding the "REST" column margin. Thus, the off-diagonal cell counts in the "REST" row, representing instances where one of the six actual subcat frames was misidentified as "REST", are counted as errors. This formula, in general, gives higher error rates than would result from simply dividing the off-diagonal cell counts by the total cell counts.

Overall, the most frequent source of errors, again, was errors in noun phrase boundary detection. The second most frequent source was misidentification of infinitival 'purpose' clauses, as in "he used a crowbar to open the door". "To open the door" is a 'purpose' adjunct modifying either the verb phrase "used a crowbar" or the main clause "he used a crowbar". But such adjuncts are incorrectly judged to be complements of their main verbs by the subcat frame extraction rules in Table 2. In formulating the rules, we assumed that a 'purpose' adjunct appears effectively randomly and much less frequently than infinitival complements. This is true for our corpus in general; but some verbs, such as "use" and "need", appear relatively frequently with 'purpose' infinitivals. In addition to errors from parsing and 'purpose' infinitives, we observed several other, less frequent types of errors. These, too, pattern with specific verbs and do not occur randomly across verbs.

4 Statistical Analysis

For most of the verbs in the experiment, our method provides a good measure of subcat frame frequencies. However, some of the verbs seem to appear in syntactic structures that cannot be captured by our inventory of subcat frames. For example, "need" is frequently used in relative clauses without relative pronouns, as in "the last thing they need". Since this kind of relative clauses cannot be captured by the rules in Table 2, each occurrence of these relative clause causes an error in measurement. It is likely that there are many other classes of verbs with distinctive syntactic preferences. If we try to add rules for each such class, it will become increasingly difficult to write rules that affect only the target class and to eliminate undesirable rule interactions.

In the following sections, we describe a statistical method which, based on a set of training samples, enables the system to learn patterns of errors and substantially increase the accuracy of estimated verb-subcat frequencies.

4.1 General Scheme

The method described in Section 2 is wholly deterministic; it depends only on one set of subcat extraction rules which serve as filters. Instead of treating the system output for each verb token as an estimated subcat frame, we can think of the output as one feature associated with the occurrence of the verb. This single feature can be combined, statistically, with other features in the corpus to yield more accurate characterizations of verb contexts and more accurate subcat-frame frequency estimates. If the other features are capturable via regular-expression rules, they can also be automatically detected in the manner described in the Section 2. For example, main verbs in relative clauses without relative pronouns may have a higher probability of having the feature "nnk", i.e., "(NP)(NP)(VERB)".

More formally, let Y be a response variable taking as its value a subcat frame. Let $X_1, X_2, \ldots, X_N$ be explanatory variables. Each X_i is associated with a feature expressed by one or a set of regular expressions. If a feature is expressed by one regular expression (R), the value of the feature is 1 if the occurrence of the verb matches R and 0 otherwise. If the feature is expressed by a set of regular expressions, its value is the label of the regular expression that the occurrence of the verb matches. The set of regular expressions in Table 2 can therefore be considered to characterize one explanatory variable whose value ranges from (NP+NP) to (REST).

Now, we assume that a training corpus is available in which all verb tokens are given along with their subcat frames. By running our system on the training corpus, we can automatically generate a $(N + 1)$-dimensional contingency table. Table 3 is an example of a 2-dimensional contingency table with X = <OUTPUT OF SYSTEM> and Y = <REAL OCCURRENCES>. Using log-linear models (Agresti 1990), we can derive fitted values of each cell in the $(N+1)$-dimensional con-

tingency table. In the case of a saturated model, in which all kinds of interaction of variables up to $(N+1)$-way interactions are included, the raw cell counts are the Maximum Likelihood solution. The fitted values are then used to estimate the subcat frame frequencies of a new corpus as follows.

First, the system is run on the new corpus to obtain an N-dimensional contingency table. This table is considered to be an $X_1 - X_2 - \cdots - X_N$ marginal table. What we are aiming at is the Y margins that represent the real subcat frame frequencies of the new corpus. Assuming that the training corpus and the new corpus are homogeneous (e.g., reflecting similar sub-domains or samples of a common domain), we estimate the Y margins using Bayes theorem on the fitted values of the training corpus by the formula given in Table 5.

$$
\begin{aligned}
&E(Y = k \mid X_1 - X_2 - \cdots - X_N \textit{ marginal table of the new corpus}) \\
&= \sum_{i_1}\sum_{i_2}\cdots\sum_{i_N} \mathcal{N}_{i_1 i_2 \cdots i_N +}\, P(Y = k \mid X_1 = i_1, X_2 = i_2, \cdots, X_N = i_N) \\
&= \sum_{i_1}\sum_{i_2}\cdots\sum_{i_N} \mathcal{N}_{i_1 i_2 \cdots i_N +}\, \frac{P(X_1 = i_1, X_2 = i_2, \cdots, X_N = i_N \mid Y = k)\, P(Y = k)}{\sum_{k'} [P(X_1 = i_1, X_2 = i_2, \cdots, X_N = i_N \mid Y = k')\, P(Y = k')]} \\
&= \sum_{i_1}\sum_{i_2}\cdots\sum_{i_N} \mathcal{N}_{i_1 i_2 \cdots i_N +}\, \frac{\mathcal{M}_{i_1 i_2 \cdots i_N k}}{\sum_{k'} \mathcal{M}_{i_1 i_2 \cdots i_N k'}}
\end{aligned}
$$

where $\mathcal{N}_{i_1 i_2 \cdots i_n +}$ is the cell count of the $X_1 - X_2 - \cdots - X_N$ marginal table of the new corpus obtained as the system output, and $\mathcal{M}_{i_1 i_2 \cdots i_N k}$ is the fitted value of the $(N+1)$-dimensional contingency table of the training corpus based on a particular loglinear model.

Table 5: Multidimensional Statistical Estimation of Subcat Frame Frequencies

4.2 Lexical Heuristics

The simplest application of the above method is to use a 2-way contingency table, as in Table 3. There are two possibilities to explore in constructing a 2-way contingency table. One is to sum up the cell counts of all the verbs in the training corpus and produce a single (large) general table. The other is to construct a table for each verb. Obviously the former approach is preferable if it works. Unfortunately, such a table is typically too general to be useful; the estimated frequencies based on it are less accurate than raw system output. This is because the sources of errors, viz., the distribution of off-diagonal cell counts of 2-way contingency tables, differ considerably from verb to verb. The latter approach is problematic if we have to make such a table for each domain. However, if we have a training corpus in one domain, and if the heuristics for each verb extracted from the training corpus are also applicable to other domains, the approach may work.

To test the latter possibility, we constructed a contingency table for the verb from the test corpus described in the Section 3 that was most problematic (least accurately estimated) among the 33 verbs—"need". Note that we are using the test corpus described in the Section 3 as a training corpus here, because we already know both the measured frequency and the hand-judged frequency of "need" which are necessary to construct a contingency table. The total occurrence of this verb was 75. To smooth the table, 0.1 is added to all the cell counts. As new test corpora, we extracted another 300,000 words of tagged text from the WSJ corpus (labeled "W3") and also three sets of 300,000 words of tagged text from the Brown corpus (labeled "B1", "B2", and "B3"),

W3	NP+NP	NP+CL	NP+INF	NP	CL	INF	REST
Measured	2.4	0.0	10.6	44.7	1.2	31.8	9.4
By Hand	0.0	0.0	0.0	69.4	0.0	30.6	0.0
Estimated	0.0	0.0	0.0	66.3	0.0	30.1	3.6
Total Occurrences: 85							

B1	NP+NP	NP+CL	NP+INF	NP	CL	INF	REST
Measured	1.8	0.9	7.9	38.6	1.8	14.9	34.2
By Hand	0.0	0.0	0.0	72.8	0.0	15.8	11.4
Estimated	0.0	0.0	0.0	76.6	0.0	14.4	9.1
Total Occurrences: 114							

B2	NP+NP	NP+CL	NP+INF	NP	CL	INF	REST
Measured	0.0	1.4	8.7	40.6	1.4	17.4	30.4
By Hand	0.0	0.0	0.0	73.9	0.0	18.8	7.2
Estimated	0.0	0.0	0.0	76.1	0.0	16.4	7.5
Total Occurrences: 69							

B3	NP+NP	NP+CL	NP+INF	NP	CL	INF	REST
Measured	3.3	0.0	1.7	30.0	3.3	31.7	30.0
By Hand	0.0	0.0	0.0	60.0	0.0	28.3	11.7
Estimated	0.0	0.0	0.0	61.4	0.0	29.8	8.8
Total Occurrences: 60							

Table 6: Statistical Estimation (Unit = %) for the Verb "Need"

as retagged under the Penn Treebank tagset. All the training and test corpora were reviewed — and judged — by hand.

Table 6 gives the frequency distributions based on the system output, hand judgement, and statistical analysis. (As before, we take the hand judgement to be the gold standard, the actual frequency of a particular frame.) After the Y margins are statistically estimated, the least estimated Y values less than 1.0 are truncated to 0. (These are considered to have appeared due to the smoothing.)

In all of the test corpora, the method gives very accurate frequency distribution estimates. Big gaps between the automatically-measured and manually-determined frequencies of "NP" and "REST" are shown to be substantially reduced through the use of statistical estimation. This result is especially encouraging because the heuristics obtained in one domain are shown to be applicable to a considerably different domain. Furthermore, by combining more feature sets and making use of multi-dimensional analysis, we can expect to obtain more accurate estimations.

5 Conclusion and Future Direction

We have demonstrated that by combining syntactic and multidimensional statistical analysis, the frequencies of verb-subcat frames can be estimated with high accuracy. Although the present system measures the frequencies of only six subcat frames, the method is general enough to be extended to many more frames. Since our current

focus is more on the estimation of the frequencies of subcat frames than on the acquisition of frames themselves, using information on subcat frames in machine-readable dictionaries to guide the frequency measurement can be an interesting direction to explore.

The traditional application of regular expressions as rules for deterministic processing has self-evident limitations since a regular grammar is not powerful enough to capture general linguistic phenomena. The statistical method we propose uses regular expressions as filters for detecting specific features of the occurrences of verbs and employs multi-dimensional analysis of the features based on loglinear models and Bayes Theorem.

We expect that by identifying other useful syntactic features we can further improve the accuracy of the frequency estimation. Such features can be regarded as characterizing the syntactic context of the verbs, quite broadly. The features need not be linked to a local verb context. For example, a regular expression such as "`w[^vex]*k`" can be used to find cases where the target verb is preceded by a relative pronoun such that there is no other finite verb or punctuation or sentence final period between the relative pronoun and the target verb.

If the syntactic structure of a sentence can be predicted using only syntactic and lexical knowledge, we can hope to estimate the subcat frame of each occurrence of a verb using the context expressed by a set of features. We thus can aim to extend and refine this method for use with general probabilistic parsing of unrestricted text.

Acknowledgements

We thank Teddy Seidenfeld, Jeremy York, and Alex Franz for their comments and discussions with us.

References

Agresti, A. (1990) *Categorical Data Analysis*. New York, NY: John Wiley and Sons.

Baker, J. (1979) "Trainable grammars for speech recognition". In D.H. Klatt and J.J. Wolf (eds.), *Speech Communication Papers for the 97th Meeting of the Acoustic Society of America*,pp. 547–550.

Brent, M.R. (1991) "Automatic acquisition of subcategorization frames from untagged text". *Proceedings of the 29th Annual Meeting of the ACL.*

Brent, M.R. and Berwick, R.C. (1991) "Automatic acquisition of subcategorization frames from tagged text". *Proceedings of the DARPA Speech and Natural Language Workshop*, Morgan Kaufmann.

Brill, E. (1992) "A simple rule-based part of speech tagger". *Proceedings of the DARPA Speech and Natural Language Workshop*, Morgan Kaufmann.

Carroll, G. and Charniak, E. (1992) "Learning probabilistic dependency grammars from labelled text". *Working Notes of the Symposium on Probabilistic Approaches to Natural Language*, AAAI Fall Symposium Series.

Church, K.W. (1988) "A stochastic parts program and noun phrase parser for unrestricted text". *Proceeding of the Second Conference on Applied Natural Language Processing.*

deMarcken, C.G. (1990) "Parsing the LOB corpus". *Proceedings of the 28th Annual Meeting of the ACL*, pp. 243–251.

Hornby, A.S. (ed.). (1989) *Oxford Advanced Learner's Dictionary of Current English*. Oxford, UK: Oxford University Press.

Jelinek, F., Lafferty, J.D., and Mercer, R.L. (1990) *Basic Method of Probabilistic Context Free Grammars*. Technical Report RC 16374 (72684), IBM, Yorktown Heights, NY 10598.

Lari, K. and Young, S.J. (1990) "The estimation of stochastic context-free grammars using the Inside-Outside algorithm". *Computer Speech and Language*, *4*, pp. 35–56.

Handling Syntactic Extra-Grammaticality

Fuliang Weng

Computer Science Department and Computing Research Lab
New Mexico State University, Las Cruces, NM 88003
email: fweng@nmsu.edu

Abstract

This paper reviews and summarizes six different types of extra-grammatical phenomena and their corresponding recovery principles at the syntactic level, and describes some techniques used to deal with four of them completely within an Extended GLR parser (EGLR). Partial solutions to the remaining two by the EGLR parser are also discussed. The EGLR has been implemented.

1 Introduction

Extragrammatical phenomena in natural languages are very common and there has been much effort devoted to dealing with them (Carbonell – Hayes, 1983; DARPA 1991, 1992). Although (Generalized)LR parsers have many merits when applied to NL, most progress with extra-grammatical phenomena has been through rule-based systems, in contrast to the applications of LR parsers in programming languages. In this paper some techniques are developed to extend the ability of a (G)LR parser in dealing with extra-grammatical phenomena, though similar techniques can also be applied in other parsers. The extended GLR (EGLR) parser is implemented.

In section 2, six types of extra-grammatical phenomena at the syntax level are classified, and then five recovery principles are introduced. Section 3 then describes the techniques used in the EGLR parser. A correctness theorem is given, and a property is also presented, showing that any sentence can be accepted if the relaxation parameter is set large enough. Section 4 gives some examples to show how the EGLR parser works. Section 5 briefly discusses the sixth type of extra-grammatical phenomenon and relevant issues, and other people's work is also compared.

2 Categorizing Extra-grammaticality at the Syntax Level

Among discussions of extra-grammaticality in natural language processing, (Carbonell — Hayes, 1983) gave a comprehensive and complete overview in this area . Here we try to rephrase these extra-grammatical phenomena from the viewpoint of purely structural possibilities at the syntactic level.

Following (Carbonell — Hayes, 1983), we use extra-grammaticality to refer to phenomena in which sequences of words are not covered by current grammar rules. In doing so, we try, at this moment, to avoid unnecessary debate about the possibility of drawing a clear line between grammatical and ungrammatical phenomena. At the end of the paper, we shall show an interesting result with a deviation degree parameter, produced by our parser that may provide some hint for understanding (un)grammaticality. In what follows, extra-grammatical phenomena contain both ungrammatical and uncovered grammatical phenomena.

The six types of syntactic extra-grammaticalities are:[1]

Phenomenon 1: the absence of a word's cate-

[1]Phenomenon 1 was discussed in (Tomita, 1985) and might be also considered to be at the lexical level. Phenomena 2, 3 and 4 and the similar recovery principles were also discussed in (Aho — Peterson, 1973; Saito — Tomita, 1991).

gories.
For example, *xyz leads arbitrariness.*, when *xyz* is not present in the English dictionary and therefore the sentence is not covered by a grammar.

Phenomenon 2: category switching.
For example, *The man on the left was talking non-sense.*, when in the dictionary entry, *left* only has a category *ADJ* but not *N*, and therefore the sentence can't be covered by a grammar with only *N* as its head in *NP* rules.

Phenomenon 3: ellipses at different levels; a category or a phrasal category is missing with respect to a rule while no other rules can cover such phenomena.
For example, *He give an apple to* _, given the only rule $PP \leftarrow P\ NP$ for *PP* formation.

Phenomenon 4: redundancy; an extra category or a phrasal category occurs with respect to a rule while no other rules can cover such phenomena.
For example, *The man lives in in a house.*, there is an extra *in*, given the only rule $PP \leftarrow P\ NP$ for PP formation.

Phenomenon 5: constituent swapping; two constituents swapped their positions with respect to a rule while no other rules can cover such phenomena.
For example, *He is happy?*, given the only rule $Q \leftarrow AUX\ NP\ VP?$ for question formation.

Phenomenon 6: non-extragrammaticality failure: a constituent is covered by rules but not intended by the speaker (or writer).
For example, *The man sit on the river bank,* given the only rule $NP \leftarrow Det\ N$ for NP formation. In this case, *the river* will form a NP, instead of *the river bank.*

We put aside the last phenomenon for a moment since it needs the coordination of multiple levels besides the syntactic one.

Corresponding to these phenomena, we propose five principles which try to remedy the failures caused by them. When a parser fails, at a point, we hypothesize several alternatives according to the following five principles, which correspond to the first five types of phenomena:

Principle 1: hypothesizing all the categories as the categories of the absent of the failed word.

Principle 2: hypothesizing the complement of all the categories of the failed word.

Principle 3: hypothesizing a pseudo word which has all the categories.

Principle 4: hypothesizing that the failed word is improperly added.

Principle 5: let xy be two consecutive words (or constituents) in the input sequence that is not parsable with respect to the current grammar, hypothesizing another word (or constituent) order, i.e., replacing xy by yx in the input sequence.

It can be easily noticed that all the principles try to *assimilate* abnormal phenomena by using known rules.

But the five principles themselves alone do not solve the problem and at least another two issues have to be dealt with:

1. to determine where the failure occurs, since most principles presuppose knowing the exact position of the failed word.
2. to determine what type of failure it is, since failure itself does not inform the type of failures the parser is encountering.

In the next section, we will present some techniques to incorporate the first four principles into an extended GLR parser, and a limited version of the fifth principle can be achieved by setting the relaxation parameter to 2, a number which will be explained later.

3 The Extended GLR Parser

The reason we choose GLR parsers as our starting point for the extensions is not arbitrary: it is closely related to the purpose of settling the two issues raised at the end of the last section. Like LR parsers, GLR parsers have the ability to detect errors in an early phase. We feel comfortable

with the first issue if we can tolerate the locality principle, i.e. the true error position is somewhere close to the place where the parser reports an error. As for the second issue, instead of deciding which type of error it is, we hypothesize all the possible error types and let all the corresponding hypotheses compete so that we can avoid answering *which type* of error beforehand. Again, GLR parsers provide a good platform for competing alternatives at different levels, although they require that comparisons be over the same input lengths.

Like GLR parsers, the extended GLR parser also has a generalized action table, a goto table and a parsing algorithm. The two tables are generated by a compiler generator, given a context-free grammar. Every entry (s, t) in the action table may either contain a set of actions (i.e., A(s, t) = $\{a_i\}$), where s stands for a particular state, t stands for a particular terminal, and a_i stands for actions which could be shifting, reducing or accepting, and A(s, t) can also be empty. The last case (i.e., A(s,t) is empty) indicates that, in state s, it is impossible to meet terminal t if the input is grammatical: in other words, the parser will report an error if, for any current state s and any category c associated with the current word, A(s, c) is empty. Like the GLR parsers when an input sentence appears, the EGLR algorithm reads one word after another from left to right, goes from one state to another, and does what the two tables specify: reduce, shift, etc. And the actions of the EGLR parser, i.e., reduce, shift and accept, are very similar to the ones in GLR parsers, except in places that will be specified later. The main differences between the GLR parsers and the EGLR parser are caused by the additional requirement of EGLR, i.e., allowing different hypotheses to compete when an error is detected (or a mismatch occurs).

The realization of different hypotheses is not so direct if we still want to stick to the following idea: allow different principles to compete at a same time, whenever it is possible, since different principles create different new sentence lengths, i.e. principles 1 and 2 do not change the length of the input sentence, principle 3 increases the length by one unit, and principle 4 reduces the length by one unit. The alignment of these length discrepancies and the compensation of its effect are realized as follows (see the next section for examples):

1. Creating a special terminal called *span*;
2. In the action table, adding an additional column which has *span* as its terminal and $S\ i$ as the value of the row of state i;
3. In the GLR parsing algorithm, the special terminals *span* along any reduce path in the graph-structured stack are ignored when a reduce action is taken.
4. Reconstructing the input sentence in the assumed error place cat_i as follows:
   ... {*span*} $\sqcup$ $\overline{cat_i}$, {*span*} $\sqcup$ cat_i, ... [2]

Among the four steps, the first three are for the parser itself and the fourth is for the input sentence. Since there might be more than one error in an input sentence, the fourth step can be repeated. We use a relaxation parameter to characterize the maximum number of times allowed in performing such relaxation.

Notice that in the fourth step, the four combinations of the two successive category sets, i.e. {*span*} {*span*}, $\overline{cat_i}$ {*span*}, {*span*} cat_i and $\overline{cat_i}$ cat_i, are exactly the first four principles [3] plus the original sentence. Since the original sentence is blocked at that point the reconstruction realizes the first four principles at the same time.

The first four principles are all in favor of local mismatches. In order to accommodate the fifth principle better, which is non-local, simple alignment is not sufficient and a more complicated mechanism needs to be used. One method for this is to use the notion of parameterization in a universal grammar. Notice that swapping two linguistic units requires the same underlying mechanism as resetting a parameter in X-bar theory (Chomsky, 1980; Gibson, 1989; Nyberg III, 1989).

Although the parser only encoded the first four principles directly, it is not difficult to see that a limited version of the fifth principle, i.e. swapping two adjacent words, is implied by the

[2] The curly parentheses indicate sets, $\sqcup$ set union, and the overline on a set refers to the set complement operation w.r.t. the set of all the terminals, *terms*.

[3] The first two principles share much in common and are dealt with in a same way.

first four principles with the relaxation parameter being 2. The details will be illustrated by an example in section 4.

One may not agree with the locality principle, i.e. that the position of real error(s) is not necessarily the place where there is no entry in the generalized action table (GAT), in other words, there are cases that can be characterized by phenomenon 6, which we will call *extra-grammatical garden path*, abbreviated as *xgp*. Obviously, *xgp* can also occur in combination with the first 5 types of phenomena. One solution to the *xgp* problem is to make some changes to cat_{i-1}. Here, we can reconstruct the input sentence at cat_{i-1} in step 4 of the alignment procedure, instead of cat_i.

It is not necessary that cat_{i-1} should be modified. In general, as a k-step *xgp*, cat_{i-k} is modified, where $k \geq 1$ and the choice of k can be based on knowledge sources at other levels, such as semantics and discourse, when $word_{i-k}$ seem to be a particularly odd fit in the local context.

Note too that step 4 in the alignment procedure can be replaced by any reasonable hypothetical sequence of categories, including the pseudo-terminal *span*. We consider the one presented here as a good candidate.

One may notice that in the phenomena and the principles enumerated, we not only allow words but also constituents to be manipulated, while up till this moment in the EGLR only words are taken as the basic units. One way to deal with constituents is as follows:

We introduce a notion called *virtual terminals*, abbreviated as *vtm*, that contain information about the corresponding non-terminals; a normal grammar is appended with a set of rules with *vtms*: $A \leftarrow *vtm_A$, where $A \in NTM$ and NTM is the set of non-terminals[4]; since $*vtm_A$ will never occur as a category of any word, the parser will not take any path containing $*vtm_A$ unless we reconstruct the input category sequence when necessary, which is very much like the case when we deal with word as unit. An example with the extended rules and their GPTs is shown in Table 3. for the set of grammar rules and its GPTs given in Tables 1 and 2.

To conclude this section, a correctness theorem and a property concerning the EGLR parser are presented as follows:[5]

> **Theorem 1.** Let $input_0 = C_1, C_2, \ldots, C_n$ be a sequence of category sets without *span*, $input_1 = C_1, C_2, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n$, and $input_2 = C_1, C_2, \ldots, C_i \sqcup \{*span*\}, \ldots, C_n$. Then a GLR parser accepts $input_0$ or $input_1$ with grammar G iff the EGLR parser accepts $input_2$ with the same grammar and relaxation parameter being 0.

> **Property:** Given any sentence and context free grammar, we can always find a value for the relaxation parameter such that the EGLR parser accepts the sentence based on the four recovery principles.

4 Some Results Produced by the EGLR

Like (G)LR parsers, the EGLR parser has two main components: a parsing table generator and a parsing algorithm. The generator takes a context-free grammar as its input and produces an action table and a goto table; while the parsing algorithm takes a sequence of words and a value of the relaxation parameter as its input and produces results, usually with four parts:

1. a sequence of triples, whose first element is the index of the triple in the sequence, whose second element is the set of the categories attached to the word in the input sentence, and whose third element is the word itself;

2. the history of the sequences of the triples, whose first element is the index of the triple in the sequence like in part 1, whose second

[4] Some non-terminals may not be able to derive a string containing only terminals, we can use a standard algorithm to detect them in (Hopcroft — Ullman, 1979) Ch 4.4, and there is no need to have the corresponding rules.

[5] The proof of the two properties are given in (Weng, 1993).

element is either the same set of the categories as in part 1 when it is parsed, or the derived one based on the alignment procedure when a mismatch happens, and whose third element is also the same as in part 1 except in the case when a mismatch occurs and then NIL is supplied;

3. (partially) successful hypothesized sequences of categories if there are mismatches, and the number of times the relaxation process is actually performed (i.e., the value of the deviation parameter);

4. a shared-packed forest representation for the sequences in item 3.

Items 2 and 3 will not be present if the original category sequence is covered by the grammar.

We now present some simple examples in order to explain better to the reader the underlying ideas, using the grammar and its corresponding GPTs given in Table 1 and Table 2.

The grammar in Table 1 does not contain any rules for pronouns. Assuming that a pronoun occurs in an input sentence and the lexicon does provide a category (*PRON) for the pronoun (i.e., a case when phenomenon 2 happens), how will the parser behave? Example sentence 1 (*They read the book.*) shows this case, where part 1 contains a sequence of sets of categories attached to the words in the sentence together with their indices in the sentence. The indices start at 0. *1-st hypothesized sentence* line in part 2 gives a guess when the parser found that the grammar does not cover *PRON, according to the principles in section 2. After feeding the new sentence to the parser internally, the parser returns the successful category sequence in part 3 and its parsing forest in part 4. A few more words about the shared-packed forest representation are given here: each row in the list is a description of a node in the forest, and it consists of two parts: the first part is the index of a node, and the second part is a tuple; the first argument of the tuple is either a terminal or a non-terminal; if it is a terminal, a T and the word[6] associated with the terminal are followed; if it is a non-terminal the follow-up lists include all its child node sets, e.g., 10 (NP (8 9)) means that node 10 is a NP having nodes 8 and 9 as its children.

```
Example sentence 1.

* (understander '(they read the book))
Part 1: the indexed sentence =
                ((0 (*PRON) (THEY 0 0))
                 (1 (*N *V) (READ 1 1))
                 (2 (*DET) (THE 2 2))
                 (3 (*N) (BOOK 3 3)))
Part 2: the history of hypothesized
sentences:
  1-st hypothesized sentence:
          ((0 (*SPAN* *V *PREP *N *DET)
              (NIL -1 0))
           (1 (*SPAN* *PRON) (THEY -1 1))
           (2 (*N *V) (READ 1 2))
           (3 (*DET) (THE 2 3))
           (4 (*N) (BOOK 3 4))
           (5 ($) (NIL NIL 5)))
Part 3:
 The following hypothesized sentences
get parsed:
(((*N (NIL -1 0)) (*V (READ 1 2))
  (*DET (THE 2 3)) (*N (BOOK 3 4))))
 and the value of the deviation
parameter is 1
Part 4:
 it is accepted !
 the root of its parse forest is (12),
and the forest is:
0   0
1   (*DET T (NIL -1 0))
2   (*N T (NIL -1 0))
3   (*SPAN* T (NIL -1 0))
4   (*SPAN* T (THEY -1 1))
5   (NP (2))
6   (*V T (READ 1 2))
7   (*N T (READ 1 2))
8   (*DET T (THE 2 3))
9   (*N T (BOOK 3 4))
10   (NP (8 9))
11   (VP (6 10))
12   (S (5 11))
the end of the forest.
NIL
*
```

Example sentence 2 shows the phenomenon 1 and the result produced by the parser. It is similar to example sentence 1, and hypotheses a set of categories for the unknown word *researchers* and produces a parsing forest for that guess.

```
Example sentence 2.

* (understander '(researchers
                  understand the book))
Part 1: the indexed sentence =
```

[6]NIL is filled if the terminal is from a hypothesized category set

```
            ((0 NIL (RESEARCHERS 0 0))
             (1 (*V) (UNDERSTAND 1 1))
             (2 (*DET) (THE 2 2))
             (3 (*N) (BOOK 3 3)))
Part 2: the history of hypothesized
sentences:
  1-st hypothesized sentence:
       ((0 (*SPAN* *V *PREP *N *DET)
           (NIL -1 0))
        (1 (*SPAN*) (RESEARCHERS -1 1))
        (2 (*V) (UNDERSTAND 1 2))
        (3 (*DET) (THE 2 3))
        (4 (*N) (BOOK 3 4))
        (5 ($) (NIL NIL 5)))
Part 3:
 The following hypothesized sentences
get parsed:
(((*N (NIL -1 0)) (*V (UNDERSTAND 1 2))
  (*DET (THE 2 3)) (*N (BOOK 3 4))))
 and the value of the deviation
parameter is 1
Part 4:
 it is accepted !
 the root of its parse forest is (11),
and the forest is:
0   0
1   (*DET T (NIL -1 0))
2   (*N T (NIL -1 0))
3   (*SPAN* T (NIL -1 0))
4   (*SPAN* T (RESEARCHERS -1 1))
5   (NP (2))
6   (*V T (UNDERSTAND 1 2))
7   (*DET T (THE 2 3))
8   (*N T (BOOK 3 4))
9   (NP (7 8))
10   (VP (6 9))
11   (S (5 10))
the end of the forest.
NIL
*
```

Example sentence 3 shows a combination of several possible phenomena. The category set sequence derived from sentence *the man lives in in the house* can not get parsed without relaxation, and the parser detects an error at the second *in* indexed as 4-th in the input sequence. Then its first hypothesized sequence is proposed and shown in part 2. Because the default value of the relaxation parameter is 1, this relaxation process is permitted and the indexed sequence is internally resubmitted to the parser. This time, it gets parsed. The parsing forest is shown in part 4 and a set of successfully hypothesized sequences are given in part 3.

```
Example sentence 3.

* (understander '(the man lives
                  in in the house))
Part 1: the indexed sentence =
          ((0 (*DET) (THE 0 0))
           (1 (*N) (MAN 1 1))
           (2 (*N *V) (LIVES 2 2))
           (3 (*PREP) (IN 3 3))
           (4 (*PREP) (IN 4 4))
           (5 (*DET) (THE 5 5))
           (6 (*N) (HOUSE 6 6)))
Part 2: the history of hypothesized
sentences:
  1-th hypothesized sentence:
          ((0 (*DET) (THE 0 0))
           (1 (*N) (MAN 1 1))
           (2 (*N *V) (LIVES 2 2))
           (3 (*PREP) (IN 3 3))
           (4 (*SPAN* *V *N *DET)
              (NIL -1 4))
           (5 (*SPAN* *PREP)
              (IN -1 5))
           (6 (*DET) (THE 5 6))
           (7 (*N) (HOUSE 6 7))
           (8 ($) (NIL NIL 8)))
Part 3:
 The following hypothesized sentences
get parsed:
(((*DET (THE 0 0)) (*N (MAN 1 1))
  (*V (LIVES 2 2)) (*PREP (IN 3 3))
  (*N (NIL -1 4)) (*PREP (IN -1 5))
  (*DET (THE 5 6)) (*N (HOUSE 6 7)))
 ((*DET (THE 0 0)) (*N (MAN 1 1))
  (*V (LIVES 2 2)) (*PREP (IN 3 3))
  (*DET (THE 5 6)) (*N (HOUSE 6 7))))
 and the value of the deviation
parameter is 1
Part 4:
 it is accepted !
 the root of its parse forest is (20),
and the forest is:
0   0
1   (*DET T (THE 0 0))
2   (*N T (MAN 1 1))
3   (NP (1 2))
4   (*V T (LIVES 2 2))
5   (*PREP T (IN 3 3))
6   (*DET T (NIL -1 4))
7   (*N T (NIL -1 4))
8   (*SPAN* T (NIL -1 4))
9   (NP (7))
10   (PP (5 9))
11   (VP (4 10))
12   (S (3 11))
13   (*PREP T (IN -1 5))
14   (*SPAN* T (IN -1 5))
15   (*DET T (THE 5 6))
16   (*N T (HOUSE 6 7))
17   (NP (15 16))
18   (PP (13 17))
19   (PP (5 17) (5 21))
20   (S (12 18) (3 22))
21   (NP (9 18))
22   (VP (4 19))
the end of the forest.
NIL
*
```

The above examples only show that if there is one extra-grammatical place in a single sentence the parser can deal with it. Actually, the ability of the parser is not limited to this. An optional relaxation parameter is offered (default is 1) by the parser. So, if there are multiple extra-grammatical places in one sentence, by setting this relaxation parameter properly the parser can still proceed with various guesses. Example 4 shows this situation. Sentence *the man home likes* is not covered by the grammar given in Table 1.

When the parser tries to parse the category set sequence derived from sentence *the man home likes* with the relaxation parameter being 1, it detects an error at the word *home* indexed as 2-nd in the input sequence. Then its first hypothesized sequence is proposed and shown in part 2. Because the value of the relaxation parameter is 1, this relaxation process is permitted and the indexed sequence is submitted internally to the parser. With the reconstruction of the input sequence, the parser proceeds and goes through word *likes*, and then it detects another error and can not go further because the relaxation parameter is 1 and no further relaxation is allowed.

A next example is about the same sentence being presented to the parser with the relaxation parameter being 2. The same process as the one in the previous example happens until the parser detects the second error at position 5. This time, another hypothesized sequence gets proposed and allowed to be internally resubmitted to the parser because the relaxation parameter is 2. It gets parsed. The parsing forest is shown in part 4.

The two successfully hypothesized sequences in part 3 need a little bit more explanation. The first sequence ((*DET THE) (*N MAN) (*V LIKES) (*N NIL)) is created as follows: after the parser parsed ((*DET) THE) and ((*N) MAN), it meets ((*ADJ *ADV *N) HOME) and detects an error as described earlier. The parser then uses the four principles to create hypotheses, and after the parser processes ((*V) LIKES), the deleting principle survives. Since the next symbol in the input sequence is the end of the sequence and the parser detects another error. Further hypotheses are made and the one with *N finally survives. Back linking the *N with the one deleted (i.e., HOME), we actually could realize a limited version of the fifth principle, i.e., swapping two adjacent words, as mentioned before. A finer degree of relaxation related to the fifth principle can also be classified as below, although its usefulness may likely be finally decided through multiple knowledge interaction:

1. When the deleted category set (e.g., cat(HOME) in the current example) has no intersection with the inserted category set (e.g., (*N) in the current example), rank it as bad; the relaxation may not be allowed in this case;[7]
2. When the deleted category set has intersection with the inserted category set, but not inclusion, rank it as acceptable;
3. When the deleted category set is contained the inserted category set, rank it as good;
4. When the deleted category set contains the inserted category set, rank it as good;

The second hypothesized sequence is similar to the first one. The difference between the first hypothesized sequence and the second one is that in the first relaxation, instead of deleting principle, the inserting principle survives. All the rest of the two hypothesized sequences are more or less the same and we are not going to explain further.

```
Example sentence 4.

* (understander '(the man
                  home likes))
Part 1: the indexed sentence =
      ((0 (*DET) (THE 0 0))
       (1 (*N) (MAN 1 1))
       (2 (*ADJ *ADV *N) (HOME 2 2))
       (3 (*V) (LIKES 3 3)))
Part 2: the history of hypothesized
sentences:
  1-th hypothesized sentence:
   ((0 (*DET) (THE 0 0))
    (1 (*N) (MAN 1 1))
    (2 (*SPAN* *V *PREP *DET) (NIL -1 2))
    (3 (*SPAN* *ADJ *ADV *N) (HOME -1 3))
    (4 (*V) (LIKES 3 4))
    (5 ($) (NIL NIL 5)))
Part 3:
 and the value of the deviation
parameter is 1
Part 4:
 the grammar does not accept this sentence!
 here is a partial result
```

[7] If the deleted category set is empty or contains only unrecognized categories, the ranking may be different.

```
0   0
1   (*DET T (THE 0 0))
2   (*N T (MAN 1 1))
3   (NP (1 2))
4   (*PREP T (NIL -1 2))
5   (*V T (NIL -1 2))
6   (*SPAN* T (NIL -1 2))
7   (*N T (HOME -1 3))
8   (*SPAN* T (HOME -1 3))
9   (NP (7))
10   (PP (4 9))
11   (NP (3 10) (1 2))
12   (*V T (LIKES 3 4))
the end of the forest.
NIL
* (understander '(the man
                  home likes) 2)
Part 1: the indexed sentence =
     ((0 (*DET) (THE 0 0))
      (1 (*N) (MAN 1 1))
      (2 (*ADJ *ADV *N) (HOME 2 2))
      (3 (*V) (LIKES 3 3)))
Part 2: the history of hypothesized
sentences:
  1-th hypothesized sentence:
   ((0 (*DET) (THE 0 0))
    (1 (*N) (MAN 1 1))
    (2 (*SPAN* *V *PREP *DET) (NIL -1 2))
    (3 (*SPAN* *ADJ *ADV *N) (HOME -1 3))
    (4 (*V) (LIKES 3 4))
    (5 ($) (NIL NIL 5)))
  2-th hypothesized sentence:
   ((0 (*DET) (THE 0 0))
    (1 (*N) (MAN 1 1))
    (2 (*SPAN* *V *PREP *DET) (NIL -1 2))
    (3 (*SPAN* *ADJ *ADV *N) (HOME -1 3))
    (4 (*V) (LIKES 3 4))
    (5 (*SPAN* *V *PREP *N *DET)
       (NIL -1 5))
    (6 ($) (NIL NIL 6)))
Part 3:
 The following hypothesized sentences
get parsed:
(((*DET (THE 0 0)) (*N (MAN 1 1))
  (*V (LIKES 3 4)) (*N (NIL -1 5)))
 ((*DET (THE 0 0)) (*N (MAN 1 1))
  (*PREP (NIL -1 2)) (*N (HOME -1 3))
  (*V (LIKES 3 4)) (*N (NIL -1 5))))
 and the value of the deviation
parameter is 2
Part 4:
 it is accepted !
 the root of its parse forest is (19),
and the forest is:
0   0
1   (*DET T (THE 0 0))
2   (*N T (MAN 1 1))
3   (NP (1 2))
4   (*PREP T (NIL -1 2))
5   (*V T (NIL -1 2))
6   (*SPAN* T (NIL -1 2))
7   (*N T (HOME -1 3))
8   (*SPAN* T (HOME -1 3))
9   (NP (7))
10   (PP (4 9))
11   (NP (3 10) (1 2))
12   (*V T (LIKES 3 4))
13   (*DET T (NIL -1 5))
14   (*N T (NIL -1 5))
15   (*PREP T (NIL -1 5))
16   (*SPAN* T (NIL -1 5))
17   (NP (14))
18   (VP (12 17))
19   (S (3 18) (11 18))
the end of the forest.
NIL
*
```

5 Future Work and Conclusions

As we mentioned above, it is not always sufficient by syntactic knowledge alone to determine which k we should take in *xgp* phenomena, i.e. knowledge sources at other levels are needed in identifying which word(s) contribute to the inconsistency of the local context and/or multi-sentential (global) context. It is further discussed in (Weng, 1993) how to identify a proper k and narrow down the number of alternative interpretations while building up the syntactic-semantic structures, based on preference Semantics (Wilks, 1975; Fass — Wilks, 1983; Slater — Wilks, 1990).

It is quite interesting to contrast one of the conclusions made in (Carbonell — Hayes, 83), i.e. error correction in compiled version parsers is not flexible, with what we have done here. We are not claiming that our work completely invalidates that conclusion, but at least we see some promise in that direction. (Malone — Felshin, 1989; Moore — Dowding, 1991) also express the feeling that it is not easy to change GLR parsers to perform relaxation, while our modification is quite moderate although somewhat tricky.

(Aho — Peterson, 1973) describes an algorithm that parses any input string to completion finding the fewest possible number of errors, by changing the grammar. The algorithm does not require the locality principle. A similar effect can be realized by trying one relaxation for every word parsed when an error message is signalled.

The last point we would like to make concerns the relationship between the criteria for gram-

maticality and the deviation/relaxation parameter. The hints provided by our parser seem to suggest that grammaticality is a graded and individual-related notion, i.e. it depends not only on the individual's knowledge about language but also the ease of absorption of certain phenomena into her or his knowledge.

Acknowledgments

The author would like to express his thanks to Dr. Y. Wilks for his helpful comments on the content and presentation of this paper. Thanks also goes to the anonymous referees for valuable comments.

References

Aho A. — T. Peterson (1972) *A Minimum Distance Error-Correcting Parser for Context-Free Languages*, SIAM J. Computing, Vol.1, No.4.

Aho A. — J. Ullman (1977) *Principles of Compiler Design*, Reading (MA): Addison-Wesley.

——— (1986) *Compiler: Principles, Techniques, and Tools*, Reading (MA): Addison-Wesley.

Berwick R. (1985) *The Acquisition of Syntactic Knowledge*, Cambridge (MA): MIT Press.

Carbonell J. — P. Hayes (1983) *Recovery Strategies for Parsing Extragrammatical Language*, American Journal of Computational Linguistics, Vol 9 (3-4).

Chomsky N. (1980) *Lectures on Government and Binding*, Dordrecht: Foris Publications.

DARPA (1991) Workshop on Speech and Natural Language Workshop.

DARPA (1992) Workshop on Speech and Natural Language Workshop.

Fass D. — Y. Wilks (1983) *Preference Semantics, Ill-Formedness, and Metaphor*, American Journal of Computational Linguistics, Vol 9 (3-4).

Fong S. — R. Berwick (1989) *The Computational Implementation of Principle-Based Parsers*, 1st International Workshop on Parsing Technologies.

Gibson E. (1989) *Parsing with Principles: Predicting a Phrasal Node Before Its Head Appears*, 1st International Workshop on Parsing Technologies.

Hopcroft J. — J. Ullman (1979) Introduction to Automata Theory, Languages, and Computation, Reading (MA): Addison-Wesley.

Leung H. — D. Wotschke (manuscript) *Tradeoffs in Economy of Description in Parsing.*

Malone S. — Sue Felshin (1989) *An Efficient Method for Parsing Erroneous Input*, 1st International Workshop on Parsing Technologies.

McRoy S. — G. Hirst (1990) *Race-Based Parsing and Syntactic Disambiguation*, in Cognitive Science 14.

Moore R. — J. Dowding (1991) *Efficient Bottom-Up Parsing*, DARPA Speech and Natural Language Workshop.

Nyberg E. 3rd (1989) *Weight Propagation and Parameter Setting*, Ph.D Thesis Proposal, Department of Philosophy, CMU.

Passonneau R. et al., (1990) *Integrating Natural Language Processing and Knowledge Based Processing*, AAAI-90.

Piaget J. — B. Inhelder (1941) *Le Développement des Quantités chez L'enfant*, Neuchatel: Delachaux it Niestle. (A Translated Chinese Version).

Saito H. — M. Tomita (1991) *GLR Parsing for Noisy Input*, in *Generalized LR Parsing*, M. Tomita (ed.), Boston: Kluwer Academic Publishers.

Slater B. — Y. Wilks (1990) *PREMO: Parsing by conspicuous lexical consumption.*

Stallard D. — R. Bobrow (1992) *Fragment Processing in the DELPHI System*, DARPA Workshop on Speech and Natural Language Workshop.

Tomita M. (1984) *Disambiguation by Asking*, COLING-84.

——— (1985) *Efficient Parsing for Natural Language*, Kluwer Academic Publishers.

——— (1987) *An Efficient Augmented-Context-Free Parsing Algorithm.*, Computational Linguistics, 13, 31-46.

Weng F. (1993) *Handling Extra-Grammaticality at the Syntactic Level in Natural Language Processing*, Master Thesis, Computer Science Department and Computing Research Lab, New Mexico State University, Las Cruces, New Mexico, U.S.A.

Wilks Y. (1975) *A Preferential Pattern-Seeking Semantics for Natural Language Inference.* Artificial Intelligence 6: 53-74.

Table 1: A Set of Grammar Rules.

Rule-id	Rule form used in EGLR	CFG Rules (sorted)
0-th	(NP → (*DET *N))	NP → *DET *N
1-th	(NP → (*N))	NP → *N
2-th	(NP → (NP PP))	NP → NP PP
3-th	(PP → (*PREP NP))	PP → *PREP NP
4-th	(S → (NP VP))	S → NP VP
5-th	(S → (S PP))	S → S PP
6-th	(START → (S))	START → S
7-th	(VP → (*V NP))	VP → *V NP
8-th	(VP → (*V PP))	VP → *V PP

Table 2: The GAT and goto Table

the generalized action table:

state-id	*DET	*N	*PREP	*V	$
0	s3	s4			
1			s7	s8	
2			s7		a
3		s10			
4			r1	r1	r1
5			r4		r4
6			r2	r2	r2
7	s3	s4			
8	s3	s4	s7		
9			r5		r5
10			r0	r0	r0
11			r3 s7	r3	r3
12			r8		r8
13			r7 s7		r7

the goto table:

state-id	NP	PP	S	VP
0	1		2	
1		6		5
2		9		
3				
4				
5				
6				
7	11			
8	13	12		
9				
10				
11		6		
12				
13		6		

Table 3: The Extended GAT.

the extended generalized action table:

state id	*DET	*N	*PREP	*V	$	*SPAN*
0	s3	s4				s0
1			s7	s8		s1
2			s7		a	s2
3		s10				s3
4			r1	r1	r1	s4
5			r4		r4	s5
6			r2	r2	r2	s6
7	s3	s4				s7
8	s3	s4	s7			s8
9			r5		r5	s9
10			r0	r0	r0	s10
11			r3 s7	r3	r3	s11
12			r8		r8	s12
13			r7 s7		r7	s13

the goto table:

state id	NP	PP	S	VP
0	1		2	
1		6		5
2		9		
3				
4				
5				
6				
7	11			
8	13	12		
9				
10				
11		6		
12				
13		6		

Table 4: The Extended Grammar.

the extended grammar:

Rule-id	Rule form used in EGLR	CFG Rules (sorted)
0-th	(NP → (*DET *N))	NP → *DET *N
1-th	(NP → (*N))	NP → *N
2-th	(NP → (*VTMNP))	NP → *VTMNP
3-th	(NP → (NP PP))	NP → NP PP
4-th	(PP → (*PREP NP))	PP → *PREP NP
5-th	(PP → (*VTMPP))	PP → *VTMPP
6-th	(S → (*VTMS))	S → *VTMS
7-th	(S → (NP VP))	S → NP VP
8-th	(S → (S PP))	S → S PP
9-th	(START → (S))	START → S
10-th	(VP → (*V NP))	VP → *V NP
11-th	(VP → (*V PP))	VP → *V PP
12-th	(VP → (*VTMVP))	VP → *VTMVP

Table 5: The Extended GPTs.

the extended generalized action table:

stt-id	*DET	*N	*PREP	*V	$	*SPAN*	*VTMNP	*VTMPP	*VTMS	*VTMVP
0	s3	s4				s0	s5		s6	
1			s9	s11		s1		s10		s12
2			s9		a	s2		s10		
3		s14				s3				
4			r1	r1	r1	s4		r1		r1
5			r2	r2	r2	s5		r2		r2
6			r6		r6	s6		r6		
7			r7		r7	s7		r7		
8			r3	r3	r3	s8		r3		r3
9	s3	s4				s9	s5			
10			r5	r5	r5	s10		r5		r5
11	s3	s4	s9			s11	s5	s10		
12			r12		r12	s12		r12		
13			r8		r8	s13		r13		
14			r0	r0	r0	s14		r0		r0
15			r4 s9	r4	r4	s15		r4 s10		r4
16			r11		r11	s16		r11		
17			r10 s9		r10	s17		r10 s10		

the goto table:

state id	NP	PP	S	VP
0	1		2	
1		8		7
2		13		
3				
4				
5				
6				
7				
8				
9	15			
10				
11	17	16		
12				
13				
14				
15		8		
16				
17		8		

Adventures in Multi-dimensional Parsing: Cycles and Disorders

Kent Wittenburg

Bellcore, 445 South St., MRE 2A-347,
Morristown, NJ 07962-1910, USA
email: kentw@bellcore.com

Abstract

Among the proposals for multidimensional grammars is a family of constraint-based grammatical frameworks, including Relational Grammars. In Relational languages, expressions are formally defined as a set of relations whose tuples are taken from an indexed set of symbols. Both bottom-up parsing and Earley-style parsing algorithms have previously been proposed for different classes of Relational languages. The Relational language class for Earley style parsing in Wittenburg (1992a) requires that each relation be a partial order. However, in some real-world domains, the relations do not naturally conform to these restrictions. In this paper I discuss motivations and methods for predictive, Earley-style parsing of multidimensional languages when the relations involved do not necessarily yield an ordering, e.g., when the relations are symmetric and/or nontransitive. The solution involves guaranteeing that a single initial start position for parsing can be associated with any member of the input set. The domains in which these issues are discussed involve incremental parsing in interfaces and off-line verification of multidimensional data.

1 Introduction

Relational Languages, those sets of expressions that are generable (or recognizable) by Relational Grammars, are characterized as relations on sets of symbols or, in practice, structured objects (Crimi et al., 1991; Golin — Reiss, 1990; Helm — Marriott, 1986, 1990; Wittenburg et al., 1991; Wittenburg, 1992a, 1992b, 1993). Sentential forms and elements of derivations are formally defined as sets of relations, each of which in turn is a set of ordered tuples, in the symbol (or object type) vocabulary set. This approach to defining multidimensional languages is compatible with unification-based approaches since the relations can be defined in unification-based grammars as constraints. Constraint Logic Programming or other approaches that enhance the usual notion of equality as the only structural constraint on terms can be employed in order to express the more free-form constraints required in multidimensional languages (see Crimi et al., 1991; Helm — Marriott, 1986, 1990; Wittenburg et al., 1991; Wittenburg, 1993). This approach to grammar and language definition generalizes over many other proposals for multidimensional grammars in the literature since arrays, graphs, and specialized spatial data structures can easily be modeled as sets of relational tuples.

Various parsers have been previously proposed for different classes of Relational languages. Bottom-up algorithms (Golin, 1991; Wittenburg et al., 1991) are the most straightforward and general, although they they are not suited to all applications. More efficient deterministic techniques proposed for graph grammars by Flasinski (1988; 1989) have been adapted by Ferrucci et al. (1991) to a constraint-based grammar framework, but the restrictions on relations and grammar productions make it unclear that the class of languages is widely useful. On the other hand, this algorithm has been shown to have a low polynomial bound on complexity. An Earley-style algorithm (Earley, 1970) has been proposed by

Wittenburg (1992a) for a larger class of nondeterministic languages whose relations are partial orders.

The limitations of these previously proposed algorithms have become evident in two domains for Relational Grammars under current investigation: on-line incremental parsing of visual language expressions (Wittenburg et al., 1991; Weitzman and Wittenburg, 1993) and off-line verification of multidimensional data. While bottom-up parsing may be employed, there are reasons to consider predictive parsing techiques. In the case of incremental parsing of visual languages consisting of connected diagrams or geometric layouts, one might use predictive parsing to detect errors as soon as they occur or to offer incremental directives (e.g., the analogue of completion in command input) to drawing or palette selections. However, to use deterministic algorithms one must use grammars that are often not expressive enough for the constraints desired. For example, no non left- or right-unique relations are allowed with the Flasinski approach and yet geometric layouts of objects of different sizes invariably include relations that, say, have two smaller objects both in some *below* relation to a larger object. The Earley-style algorithm of Wittenburg (1992a) is unsuited to incremental interface applications because more flexibility in a parser's scanning order is desired than is afforded by this algorithm. Unlike with one-dimensional text, it is not easy to anticipate in a two- or n-dimensional world exactly what orderings will seem most natural to users. It seems clear that for most diagramming languages, for example, interfaces that allow a good deal of flexibility in how users build up diagrams would be preferred over interfaces that insist that users build up diagrams in prespecified orders that happen to conform, say, to temporal control relations that are reflected in the arcs of the diagrams. Further, many visual languages contain relations that may be symmetric or nontransitive, in which case the Wittenburg (1992a) algorithm is not usable. One set of examples arises with languages whose primitives are directed line segments. Relations between line segments such as head-to-head or tail-to-tail are symmetric, precluding the use of algorithms depending on the intrinsic ordering properties of the relations to direct scanning. Cycles are of course generally common in flowchart diagramming languages to represent control loops.

In the case of off-line data verification and correction, the size of the input sets involved may preclude bottom-up parsing for efficiency reasons. The following scenario from a current Bellcore application domain is an example of the problem. Suppose one is verifying that all the line segments that comprise the border of each region in a mapping database in fact enclose that region and that any attributes of the data are in conformance. One approach is to define grammars that independently combine the line segments surrounding each region to yield a closed polygon of some form. But blind bottom-up parsing would take each and every line segment to be the start of the polygon constituent, rebuilding it many, many times over. In geographical data sets such as the U.S. Census Bureau's Tiger database, the number of line segments representing the border of a single state typically reaches into the thousands. It is not hard to see that such a redundant algorithm would be ill-advised. Not only is it inefficient, but the problem of error detection is, at least on the face of it, made more difficult than if the parsing enumeration were systematically ordered and based on systematic prediction.

The goal addressed in this work is to design a predictive, Earley-style algorithm for Relational Grammars without relying on the relations themselves to provide scanning orderings. Such an algorithm allows predictive parsing methods to be employed with higher-dimensional languages whose relational graphs contain cycles, i.e., one or more of the relations are not partial orders. To design such an algorithm requires solving the problem of finding a start element for the parser to begin its predict and scan operations. The Wittenburg (1992a) algorithm uses minimal elements of the relations to initialize the parser at possibly multiple starting positions. The goal here is to allow the parser to start with a single arbitrary member of the input set and still guarantee completeness.

The remainder of this paper is structured as follows. First, as a review and refinement of work to date, a Relational Grammar formalism for practical applications developed and implemented at

Bellcore is summarized, followed by relevant aspects of Earley-style parsing from Wittenburg (1992a). We then turn to the question of how to define a suitable subclass of Relational Grammars for ensuring that parsing may be initiated from an arbitrary starting point. An Earley-style parsing algorithm and example trace are presented next. The conclusion includes remarks regarding the extensions to this work necessary to fully solve the problems of incremental, predictive parsing for higher-dimensional languages.

2 A Relational Grammar Formalism

Relational Grammars are motivated by domains in which the sets of objects to be generated or analyzed can not comfortably be represented as strict linear orders of symbols. Examples include expressions in 2-D such as mathematics notation, flowcharts, or schematic diagrams; 2-D or 3-D graphical layouts and displays, perhaps with time or additional media as added dimensions; and n-dimensional data to be found in empirical data collections or various types of databases. One must generalize the data type of language expressions from, say, one-dimensional arrays (strings) to 2-, or perhaps n-dimensional arrays or, more generally, to graphs of relations where the nodes represent data in unrestricted formats. Notions of replacement in derivations have to be generalized also, and, as attested by the literature on array and graph grammars, there are many variations on how to define grammar productions and the notion of replacement (called the embedding problem in the graph-grammar literature).

The Relational Language (RL) framework provides for an abstraction over the particular data structures used to hold the object sets being processed. The grammar productions make mention of the relations expected to hold in the data but there is flexibility in choosing exactly how the data is represented and stored as well as the implementation of how the relations are checked or queried. Such an approach accommodates many kinds of data representations, including array or graph structures, K-D trees, or even commercial databases in which indexings may be precisely tuned for efficiency.

Combining input objects during the parsing process can be characterized neutrally through set operations — set union then being a very general analog of string concatenation. It is natural to think of a grammar rule as providing a definition of a composite (nonterminal) object as a set (usually nonunary) whose type is the symbol on the left-hand-side of the rule and whose parts are the union of the parts of the objects whose types are the symbols on the right-hand-side of the rule. Derivations are defined as a sequence of replacements that are headed by a type that is a root symbol of the grammar and terminate in a set of objects whose types are taken from the vocabulary of terminal symbols. An important characteristic of the RL approach is that derivations are trees, as is the case in conventional context-free grammars. An effect of this restriction is that no object may be used more than once per derivation.

The most significant new requirement for the grammar formalism is that it has to provide a means for specifying possible combining relations explicitly. Specifying combining relations in Relational Grammars is done by stating, for each element in the right-hand-side of a non-unary rule, what relation it has to stand in with respect to at least one other right-hand-side element. Operationally, a parser finds relevant input for combination by executing queries formed from the relational constraints.

Besides the unification-based approaches to Relational Languages mentioned above, a more efficient grammar compiler has also been implemented (Wittenburg, 1992b) that incorporates a form of "pseudo-unification" (see Tomita, 1990). One may consider the pseudo-unification-based formalism as syntactic sugar for an underlying unification system. From this point of view, the results of this paper generalize to the full family of extended unification-based approaches and thus it is not the case that we are dealing here with yet another special-case grammar formalism. Nevertheless, we will use the pseudo-unification-based formalism in this dicussion since it is less verbose than the full unification-based specification. It also of course affords the possibility that specialized algorithms may be found that can be proved more efficient than general unification.

Example 1.

```
(defrule (Subtree-rule example-grammar)
    (0 Subtree)
    (1 prim)
    (2 Row)
    :expanders (below 2 1)
    :predicates (centered-in-x 1 2))
```

Example 1 shows a defining form for a simple rule that states that an object of type Subtree can be composed out of two objects of type prim and Row, as long as they stand in the stated *below* and *centered-in-x* relations. The integers in the textual rule definition act as references to rule elements: the left-hand-side of a rule is conventionally marked as 0; the one or more right-hand-side elements are numbered 1...n. The backbone of this rule thus could be written as Subtree → prim Row. A relational constraint such as (below 2 1) is to be interpreted as a requirement that the object matching rule element 2 (of type Row in this case) must stand in the *below* relation to the object matching rule element 1 (of type prim).

During parsing, relational constraints either have the effect of generating possible candidates for rule element matches or filtering candidate matches that have been proposed. Relational constraints immediately following the keyword :expanders act as a generators. These relations must be binary, and the parser will execute a query based on these relational expressions as it explores candidates to match rule elements. For example, the query (below :? i) would be executed when matching rule Subtree-rule in order to expand the match to the second right-hand-side element, assuming i is an index to the input matching the first right-hand-side element. We call the binary, generating relations *expander* relations, since their main role is to expand rule matches. In addition, one may include further relational constraints (of any arity). These non-initial constraints will be executed as predicates. In Example 1, (centered-in-x 1 2) is the only predicate.

Figure 1 shows a graphical depiction of the rule in Example 1 in which composition is represented as spatial enclosure. Thus one sees that the Subtree object is composed of the prim and Row objects. The arrows represent the required spatial relations.

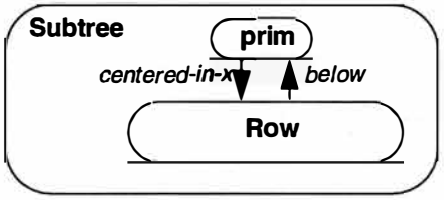


Figure 1: A simple layout rule

At a definitional level, the right-hand-side elements of RG rules are unordered — what really matters are the relational constraints, which only partially determine the order in which a parser might match the rule elements. But a parser is going to have to match rule elements in some order or other. In bottom-up parsing, it is possible to choose only a single ordering of the right-hand-side elements of each production. Wittenburg et al. (1991) discuss the constraints in choosing such an ordering. The following constraint, which we refer to as the connectedness constraint, must hold of an order for right-hand-side rule elements in RGs:

Restriction 1. For an ordering of rhs rule elements $D_1 \ldots D_n$, there must exist at least one expander constraint between each element D_j , $1 < j$, and an element D_i where $i < j$.

That is, considered as a graph with the expander relations as arcs, the right-hand-side of a rule must be connected and, when ordered, each element in turn must be connected to some other element earlier in the ordering. This requirement implies that this class of Relational Grammars can generate only connected relation graphs since, for every production, we assume that there is at least one ordering that meets this condition. Since the expander constraints are going to provide the parser with a query function that can find the candidates to combine with next at each step, this restriction ensures that an expander query can fire at each stage in the rule match. Once an ordering is found, a grammar compiler can place an expander or predicate expression with the first rule element in which all the arguments of the relational expression will be bound.[1]

[1] Ordering the constraints in a grammar compiling operation as discussed here obviates the need for residuation in unification operations, the basis for the unification extensions discussed in Wittenburg (1993).

In remaining examples in this section, we will assume that the rules are ordered as shown, and we will associate the relational constraints with the relevant rule elements rather than list them at the end of the rule definition.

A definition useful in subsequent sections on parsing follows:

Definition 1. We know, given Restriction 1, that for every ordered production, an expander constraint of the form (rel x y) must exist for every ordered daughter at position $j > 1$ where either x or y will be grounded by matching a daughter at position j and either x or y (whichever doesn't satisfy this previous condition) is already grounded by a daughter at a position $i < j$. These arguments are defined as the to-be-bound argument and the already-bound argument, respectively, at position j.

A fundamental issue in Relational Grammar representation is whether to allow nonterminals to appear as direct arguments to relational constraints. The natural interpretation of a nonterminal in such a constraint is that it is a reference to the set of input objects in its derivational yield. Let us presume a grammar having the rule in Example 1 has other rules expanding the nonterminal category Row in such a way that a bottom-up parser can build up a horizontally aligned set of primitive objects. Consider the effect of executing the query (below :? i), mentioned earlier, when parsing. If this query is executed against the original input only, it will never find a Row object since the existence of such an object is an artifact of parsing. A solution to this problem requires dynamic updating of what we call an object store, which starts out as a collection of input objects. Composite objects are introduced as the parser finds them through rule matches. A grammar thus must define derived nonterminal objects in terms of terminal objects. For example, if input data is characterized as rectangular regions (which is reasonable for many graphical applications), then composites introduced through rule matches might be defined as the summation of the rectangular regions of the rule's daughters.

Including relational constraints directly on composites is reasonable when using bottom-up parsing, but it complicates the definition of Relational Grammars as generative systems since the composition-of relation must in principle be reversible. Further, significant problems are introduced for Earley-style parsing, or any other form of predictive parsing, as discussed in Wittenburg (1992a). The alternative is to write grammars that state relational constraints only on individuals in the input set and use feature percolation to pass up bindings of these individuals as attribute values in derivations. We will refer to this latter subclass of Relational Grammars as Atomic Relational Grammars (ARGs), noting that the most significant restriction is that the arguments of relational constraints must be atomic.

As an illustration, consider Example 2, the rule set of a flowchart grammar fragment. The root symbol for this grammar is Flowchart.

Example 2: Flowchart Grammar.

```
(defrule (flowchart flowchart-grammar)
    (0 Flowchart (setf (in 0) 1
                       (out 0) 3)
    (1 oval)
    (2 P-block (connects-to 1 (in 2)))
    (3 oval (connects-to (out 2) 3))))

(defrule (conditional flowchart-grammar)
    (0 P-block (setf (in 0) 1
                     (out 0) 3))
    (1 diamond)
    (2 P-block (Y-connects-to 1 (in 2)))
    (3 circle (connects-to (out 2) 3)
              (N-connects-to 1 3)))

(defrule (basic-p-block flowchart-grammar)
    (0 P-block (setf (in 0) 1
                     (out 0) 1))
    (1 rectangle))
```

A graphical depiction of the is shown in Figure 2. A visual indication that these relations do not hold of composite sets directly can be seen as the arcs (representing relations) cross the enclosing perimeters of nonterminal objects. All relations in this example are taken as constraints on individual members of the input set. Consider, for example, the relational constraint (connects-to 1 (in 2)) appearing in rule flowchart. The first argument, 1, is a direct reference to a terminal object with lexical type oval. The second argument, (in 2), is an indirect reference to the value of the in

attribute of an object of (nonterminal) type P-block. This value will be bound to a terminal object during parsing. We call the set of attributes expander attributes that appear in any of the arguments to expander relations in a grammar. In this grammar, *in* and *out* are the expander attributes.

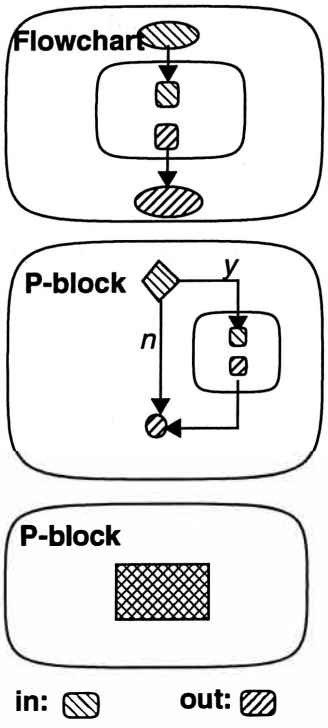


Figure 2: Graphical views of flowchart productions.

The rules must percolate references to individual members of the input as feature values from the right-hand-side of each rule to its left-hand-side. In the pseudo-unification formalism, assignments are made through setf forms. (We will also use forms in the text such as $(\text{attr}_x\ 0) = (\text{attr}_y\ i)$ to represent feature percolation. They are intended to be operationally equivalent to the setf forms.) Unlike general attribute passing, we assume equality as the only relation between values in feature percolation.[2] In the rule basic-p-block, one can see how the values of features in and out are linked directly to terminal input, in this case, an individual input object of lexical category rectangle.

We propose the following restriction for Atomic Relational Grammar productions, whose utility will become most evident when we consider predictive parsing later in this paper:

Restriction 2. Each production must percolate a value for every expander attribute used in the grammar.

In the grammar of Example 2 we can see that this condition is met since *in* and *out* are the only expander attributes used in the grammar and every production associates the value of each of these attributes in its left-hand-side with some value on its right-hand-side.

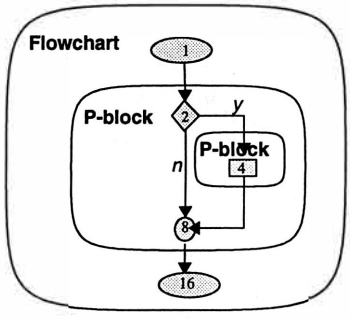


Figure 3: A derivation.

A derivation tree is shown in Figure 3. The input set is indicated as filled shapes indexed by integers representing binary numbers. Relations between input elements are graphed as arcs. The derivation tree (omitting ordering of daughter elements) is shown through the convention of spatial enclosure with dominating nonterminals represented as enclosing rounded rectangles.

3 Earley-style Parsing for Partial Orderings

Wittenburg (1992a) proposes a subclass of Atomic Relational Grammars amenable to Earley-style parsing. The class is called Fringe Relational Grammars (FRGs), where fringes are defined to be the minimal and maximal elements of Relational language expressions. The definition of Fringe Relational Grammars guarantees that any expression generable by the grammar

[2]This restriction to equality is significant only for expander attributes in percolation statements. The grammar formalism supports the use of additional features whose values may be tied to arbitrary computation using other features.

can be (partially) ordered. The motivation for such a requirement is that if we can partially order the input, then starting positions can be defined as the minimal elements and the parser can be given a partial order for scanning the input.

This is a summary of the restrictions on Atomic Relational Grammars that define the Fringe Relational Grammar subclass.

Restriction 3. Each relation used in an expander constraint in the grammar must independently be a partial order.

Restriction 4. For each expander relation used in the grammar, a pair of minimal/maximal expander attributes must be declared and every production must percolate values to these attributes in a manner that retains the partial orderings.

Restriction 5. Each production in the grammar must have an ordering variant of its right-hand-side such that, for each expander attribute, the right-hand-side element percolating the value to that expander attribute appears first.

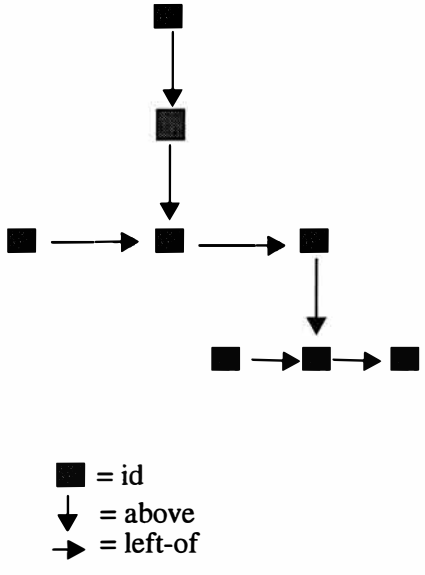


Figure 4. An expression in L(FRG-gram)

The rule set in Example 3 for the grammar we'll call FRG-gram generates Relational language expressions such as that graphed in Figure 4.

Example 3: FRG-gram rule set.

```
(defrule (S-rule FRG-gram)
    (0 S (setf (above-min 0)(above-min 1)
               (above-max 0)(above-max 1)
               (left-of-min 0)(left-of-min 1)
               (left-of-max 0)(left-of-max 1)))
    (1 Subtree))

(defrule (Subtree-rule FRG-gram)
    (0 Subtree (setf (above-min 0) 1
                     (above-max 0) (above-max 2)
                     (left-of-min 0) 1
                     (left-of-max 0) 1))
    (1 id)
    (2 Row)
    :expanders
     (above 1 (above-min 2))))

(defrule (Row-rule FRG-gram)
    (0 Row (setf (above-min 0)(above-min 2)
                 (above-max 0) (above-max 2)
                 (left-of-min 0) (left-of-min 1)
                 (left-of-max 0)(left-of-max 3)))
    (1 Subtree)
    (2 Subtree)
    (3 Subtree)
    :expanders
    (left-of (left-of-max 1)(left-of-min 2))
    (left-of (left-of-max 2)(left-of-max 3)))

(defrule (Basic-subtree FRG-gram)
    (0 Subtree (setf (above-min 0) 1
                     (above-max 0) 1
                     (left-of-min 0) 1
                     (left-of-max 0) 1))
    (1 id))
```

The *left-of* relation is used to compose horizontally aligned rows via topmost elements and *above* is used to vertically align mother elements with daughter rows via the daughter element at the row's center.

It is easy to see that both the *left-of* and *above* relations can be defined such that they independently will be partial orders on any input. This takes care of Restriction 3 for FRGs. As for the Restriction 4, note how each production sets the value of all four expander attributes *above-min*, *above-max*, *left-of-min*, and *left-of-max* in its left-hand-side category. The linkings that are defined in all cases are consistent with the partial orderings that the relations induce on the right-hand-sides of each rule. In order to meet Restriction 5,

we must spawn ordering variants for some of the rules. Table 1 summarizes such an expanded ruleset that can be generated automatically. Each ordering variant is indicated by rule name and right-hand-side sequence as defined in Example 3. From Table 1 we can see that three rule variants would need to be added over the orderings implicit in the base grammar of Example 3. Such a table is used on-line in the parsing algorithm in order to provide a mapping from expander attributes (used in prediction) to rule variants that percolate (or bind) that expander attribute first.

Table 1: Expanded FRG-gram ruleset

Rule	Expander Attribute	RHS Ordering
S-rule	*all*	<1>
Subtree-rule	above-min	<1, 2>
	above-max	<2, 1>
	left-of-min	<1, 2>
	left-of-max	<1, 2>
Row-rule	above-min	<2, 1, 3>
	above-max	<2, 1, 3>
	left-of-min	<1, 2, 3>
	left-of-max	<3, 2, 1>
Basic-subtree	*all*	<1>

The intuitive idea of the parsing algorithm is as follows. The parser is initialized in Earley style with dotted productions expanding root symbols of the grammar at each of the minimal elements of the input. Extended predict, scan, and complete steps carry over from Earley's algorithm, so the parser will build up item sets through enumerations of the nodes of the input relation graph originating at the minimal nodes. Note that no matter where the parser starts in the input, a successful derivation starting from that position will have to visit all the nodes of the input graph at least once. The main goal in designing the algorithm is to minimize the number of enumerations while still ensuring completeness. When two or more enumeration sequences converge, the algorithm is able to detect when a prediction has been made before so that the same prediction (and all items that ensue) don't have to be re-created. In order to be able to absorb the products of a previous prediction into a converging enumeration sequence, a fourth step has to be added to Earley's predict, scan, and complete. It is called inverse-complete. Where complete is given an inactive state and asked to extend active states that end where the inactive state starts, inverse-complete is given an active state and tries to extend it with any inactive states that start where the active state ends.

As suggested in the concluding remarks of Wittenburg (1992a), there is still research to be done to minimize enumeration sequences further. One strategy is look for ways of reducing the number of requisite start points. While we address a different problem in the remainder of this paper, the solution is relevant to this question since the parser need only initialize its search at a single position.

4 Preliminaries to a Parser

Here we present the preliminaries necessary to define an Earley-style algorithm for Atomic Relational Grammars that can be initialized from any start element. As is discussed in the concluding remarks, the parsing algorithm solves only part of the problem for defining incremental, predictive parsing for Relational Grammars, but it is of interest in its own right. We begin with the following observations.

The existence of an Earley predictive state (an active state that covers no input) in a parse table implies that a derivation headed by the nonterminal on the left-hand-side of the dotted rule may "begin" at that positional index. A scanning action is valid, given some positional index, only if the terminal symbol at that position is a valid left-corner of a possible derivation subtree predicted to start at that index. For a parser to adquately predict top-down expansions of the grammar's root symbol from any position in the input and only that position, it follows that every element of any parsable input sets must be a possible left-corner of a valid derivation tree headed by the root symbol. As a condition on a grammar, this implies that for every nonterminal, there must be variants of any production expanding that nonterminal such that any member of its terminal yield can be ordered first in the production's right-hand-side.

Part of our proposal then is to require order-

ing variants of the right-hand-sides of every production so that every rhs-element appears first in at least one variant. It is not hard to see that such a mechanism carried through will satisfy the any-start requirement. Intuitively, if for any local subtree in a derivation admitted by the base grammar, there exists another derivation subtree reordered such that an arbitrary rhs-element appears first, then, by induction, any derivation tree can be transformed by a series of local reorderings such that an arbitrary terminal node appears as a left-corner in at least one (other) derivation tree.

Then there is the question of remaining ordering variations once a parser has chosen a start position. Note here that it is *not* a requirement that once the parser has chosen its first element, that the next choice for scanning may arbitrarily be any of the remaining input elements. A natural requisite for ordering variants once the algorithm has been initialized at a start position is to allow for any variations of input elements that are connected through some relation to the input already scanned. Such an approach would be consistent with the parsing algorithms discussed previously since queries formed from relational constraints could serve as the basis for expanding rule matches. To fully realize even this set of ordering variations would require extensions beyond the scope of the present paper, however. Here we concentrate on the problem of an arbitrary starting point and choose to direct the parser to scan remaining input nondeterministically in orders (not necessarily all orders) consistent with the connectedness constraint just mentioned. A simple extension allows for all connected ordering variations within local rules.

In Wittenburg (1992a), the predict subroutine makes use of the function F-permute to find all candidate rules for prediction. This function maps from an expander attribute and a predicted nonterminal to rule variants appropriately ordered that can expand that nonterminal. Appropriately ordered here implies that the production can provide a possible percolation path such that, as the left-branch of an eventual derivation subtree bottoms out, terminal elements scanned at that position can ground the expander attribute used in prediction. Such a mapping is extracted from what was called an F-permute table, as exemplified in Table 1. To get an intuitive grasp of this relationship of percolation paths and prediction, which will be carried over here, consider again the input in Figure 3 with the grammar in Example 2. Assume that an Earley-style parser has scanned the topmost oval, indexed as 1 in the figure. This would imply the existence of an item that incorporated a dotted production of rule Flowchart, repeated here.

```
[flowchart-1: Flow -> start . P-block end]

    (0 Flowchart (setf (in 0) 1
                       (out 0) 3)
    (1 oval)
    (2 P-block (connects-to 1 (in 2)))
    (3 oval (connects-to (out 2) 3))
```

Note now that the expander constraint (connects-to 1 (in 2)) is of relevance in predicting the next input to be scanned. Given just this item, only those input elements that are candidates to bind the in attribute of a P-block constituent need be considered. In particular, we need not consider ordering variants of P-block rules in which the initial right-hand-side element cannot serve to bind this attribute. These observations lead us to the following definition.

Definition 2. A triple $< N, \text{attr}, P >$, N a nonterminal category, 'attr' an expander attribute, and P an ordered production of Atomic Relational Grammar $\mathcal{G}$ is in the starts-by-binding relation iff the left-hand-side category of $P = N$ and there exists a feature assignment statement of the form (attr 0) = 1 or (attr 0) = (attr′ 1) in P.

The starts-by-binding relation forms the basis for precompiling a Prediction Table. A rule variant in the table indexed by some nonterminal category X and some expander attribute attr implies that the production expands nonterminal X and that there is an assignment of the form (attr 0) = ...1... in the feature percolations of that production, i.e., the cover of the first right-hand-side element grounds attr if it is a terminal or it carries an attribute whose value is linked to attr if it is a nonterminal.

The derivation of a table representing this relation is in two steps. We first generate rule ordering variants. Here we assume an algorithm that, for each production, forms one ordering variant per right-hand-side element such that the

right-hand-side element is ordered first. Ordering the remaining elements is done arbitrarily subject to the connectedness condition in Restriction 1.[3] Step two generates a table representing the starts-by-binding relation in the grammar and, in addition, we include in the table the special "attribute" start, whose entry includes a set of rule variants such that for each production expanding that nonterminal category, there is at least one variant ordering each right-hand-side element first, regardless of feature percolations.

Example 4 shows the Atomic Relational Grammar appearing previously in Example 2 with the addition of named ordering variants for non-unary productions.

Example 4: Extended Flowchart grammar.

```
(defbaserule (flowchart flowchart-grammar)
    (0 Flowcart (setf (in 0) (in 1)
                      (out 0) (out 3)))
    (1 start)
    (2 P-block)
    (3 end)
    :expanders
    (connects-to (out 1) (in 2))
        (connects-to (out 2) (in 3)))

    Ordering variants:
    flowchart-1 <1, 2, 3>
    flowchart-2 <3, 2, 1>
    flowchart-3 <2, 1, 3>

(defbaserule (conditional flowchart-grammar)
    (0 P-block (setf (in 0) (in 1)
                     (out 0) (out 3)))
    (1 decision)
    (2 P-block )
    (3 junction)
    :expanders
    (Y-connects-to (out 1) (in 2))
    (connects-to (out 2) (in 3))
    :predicates
    (N-connects-to (out 1) (in 3)))

    Ordering variants:
    conditional-1 <1, 2, 3>
    conditional-2 <3, 2, 1>
    conditional-3 <2, 1, 3>

(defrule (basic-p-block flowchart-grammar)
     (0 P-block (setf (in 0) (in 1)
                      (out 0) (out 1)))
     (1 procedure))
```

Table 2: Flowchart-grammar Prediction Table

Nonterm	Expander Attribute	Production Variant
Flow	in	flowchart-1
	out	flowchart-2
	start	flowchart-1 flowchart-2 flowchart-3
P-block	in	conditional-1 basic-p-block
	out	conditional-2 basic-p-block
	start	conditional-1 conditional-2 conditional-3 basic-p-block

Table 2 shows a Prediction Table for the grammar in Example 4. We assume the existence of a function starts-by-binding(N, A) that for some Atomic Relational Grammar $\mathcal{G}$ returns the set of production variants in the Prediction Table at nonterminal N and attribute A.

Definition 3. The first-attrs of an ordered production p in Atomic Relational Grammar $\mathcal{G}$ is $\{\text{attr}_x\}|$ there exists an assignment statement $(\text{attr}_y\ 0) = (\ \text{attr}_x\ 1)$ in p where $\text{attr}_{x,y}$ are expander attributes.

For example, the first-attrs of ordered production conditional-1 is the set {in}. The first-attrs of ordered production basic-p-block is the set {in out}. The first-attrs of a production are those expander attributes associated with the first rhs element of a production that provide bindings for expander attributes in the left-hand-side. They are used in the recursive predict step of the Earley-style algorithm to follow.

5 Earley-style Recognition for ARGs

We now turn to an Earley-style algorithm for the full class of Atomic Relational Grammars in

[3] If a scanning algorithm is desired that will allow for all orderings consistent with Restriction 1, then more variants can simply be produced here.

which the starting position is arbitrary. We will summarize the essential points here, and confine our attention to a recognition algorithm.

Definition 4. A multidimensional multiset (md-set) is an n-tuple $(I, R_1 \ldots R_n)$ such that $R_1 \ldots R_n$ are binary relations on the multiset I.

Even though $R_1 \ldots R_n$ may in general be n-ary, expander relations must be binary and that is all we will concern ourselves with here.

Definition 5. An indexed md-set C is an md-set $(I, R_1 \ldots R_n)$ and a one-to-one and onto function from the set of integers $1 \ldots |I|$ to members of I.

Now we define the states, or items, used in a parse table given an indexed md-set C. Inactive states are representative of completely matched rules and thus they include a category as well as the feature values assocated with the rule's left-hand-side.

Definition 6. Inactive states relative to an indexed md-set C are a triple $[cat, f, c]$ where cat is a nonterminal or terminal symbol, f is a vector of features (especially, expander attributes with values in C), and c is a logical bit vector representing a subset of C (the state's terminal yield, or cover).

Inactive states will be indexed in the parse table by every binding of an expander attribute in f (all of which must be individuals in C). Intuitively, we consider inactive states to begin as well as end at every terminal that is bound by an expander attribute. However, this parser doesn't make the distinction between beginning and ending so we need only a single indexing array.

We next turn to active states, which represent partially matched or unmatched rules. We assume the Earley algorithm convention of a set of dotted productions, with the dot representing a position in the ordered right-hand-side elements.

Definition 7. Active states are a triple $[p, i, (d_1 \ldots d_n)]$ where p is a production; i, the Earley positional dot, is an integer ranging from 1 to the length of the right-hand-side of p representing the next daughter to match; and $(d_1 \ldots d_n)$ is an ordered list of pointers to inactive states of daughters matched so far.

The cover, or terminal yield, of an active state is derivable from the covers of the inactive states that have already been matched.

Definition 8. The cover of an active state $[p, i, (d_1 \ldots d_n)]$ is defined to be the union of the covers of the inactive states $(d_1 \ldots d_n)$.

As with inactive states, active states are indexed by individual members of C. The intuition is that active states are indexed by individuals in the input that are candidates to be used in the next advancement of that active state. For any daughters but the first, one can make use of the expander constraint at that positional dot to find such candidates. For active states that have not yet matched any daughters, their input indices are derived from higher predictions, initialized with the input element at the starting position.[4]

The following definition is useful in the recursive predict step of the Earley-style algorithm. It is necessary to pick out the expander attributes that can acquire bindings through matching the next daughter.

Definition 9. The to-be-bound-attrs of an active state $s = [p, i, (d_1 \ldots d_n)]$ are defined to be first-attrs(p) if $i = 1$, else $\{\text{attr}_x\}$ where attr_x is the attribute of the to-be-bound argument of the expander constraint at position i.

Agenda items are defined next.

Definition 10. An agenda item is a pair [state, keys] where state is an active or inactive state and keys is a set of state indices (individuals in an indexed multidimensional set C).

[4] In Wittenburg et al. (1991), active states were indexed by the already-bound argument. The change is necessary to make indexing of predictive states (which have no cover, and thus no bound argument) consistent with the indexing of other active states.

Agenda items represent states and their indices relative to some input. As in chart parsing, the agenda will hold a list of items to be potentially added to the parse table. There is flexibility in its management. Here we assume a FIFO queue.

Procedure 1. `Advance(a-state, i-state, a-index)`

Input: An active-state a-state = $[p, i, (d_1, \ldots d_n)]$, and an inactive-state i-state = $[\text{cat}, f, c]$.[5]

Output: A list of agenda items, possibly null, that are the result of advancing `a-state` with `i-state`.

Method:

Case 1: If i = length of rhs of p (new state will be inactive), then let c' = union-covers($(d_1 \ldots d_n), c$) and let f' = percolate($p, (d_1 \ldots d_n), c, f$). Return an agenda item $[\text{i} - \text{state}' = [\text{cat}', f', c'], \text{keys}]$ where cat′ = lhs of p and keys is the list of inactive state indices of i-state′.

Case 2: (New state will be active.) Let a-state′ = $[p, i+1, c' = (d_1, \ldots d_n, c)]$. Let e = expander-at-position($p, i+1$). Let q = query(expander, c', Q). If q is non-null, return an agenda item $[\text{a} - \text{state}', q]$.

The advance procedure is called by scan, complete, and inverse-complete, to be defined shortly. As we have pointed out, active states are indexed at an element in the terminal yield of any potential next daughters to be matched (rather than the last one to have been matched), so they will be added to the parse table only if some tuple in the required relation can be shown to exist in the input. This question is satisfied by the query(expander, c', Q) form. Here we assume that given an expander constraint and the daughters matched so far, a subroutine can dereference the arguments to the expander constraint, binding the already-bound one, and then execute a query that will return the members of the input that can bind the to-be-bound argument.

Algorithm 1. Membership in L(ARG)

Input: An Atomic Relational Grammar = $\mathcal{G}$, an indexed multidimensional set $\mathcal{C}$, and a starting element q that is an arbitrary member of $\mathcal{C}$.

Output: A parse table of state sets S_j.

Auxiliary data structures:

Agenda: a FIFO list of states to process, initially null.

Init-states: the set of starting predictive states, initialized as follows: For every p in starts-by-binding(S, start), add a state $[p, 1, \text{null}]$ to init-states. For every state $s = [p', 1, \text{null}]$ in init-states, if the rhs symbol X at rhs position 1 of p' is a nonterminal, then let init-states = union(init-states, starts-by-binding(X, start)).

Parse table: a hash table of state sets S_j, where j is an index to individuals in $\mathcal{C}$.

Method: We initialize as follows:

- For every s in init-states, add an agenda item $[s, \{q\}]$ to the Agenda.
- Until Agenda is empty, do:
  - Remove one item = [state, keys] from Agenda.
  - For k in keys, if an equivalent state is not already at S_k, add state at S_k. Then do:

    Scanner: If state = $[p, i, (d_1 \ldots d_n)]$ at k is active and the rhs symbol x at position i of p is terminal, if the terminal symbol of input item = y at k matches x and k does not intersect cover(state), then add any item in advance(state, y, k) to Agenda.

[5] A variant of this procedure must also accommodate input items from C directly in place of the inactive-state argument when called by the scan procedure. It is straightforward to form a (transient) inactive state from an input item.

Predictor: If state = $[p, i, (d_1 \ldots d_n)]$ is active and the rhs symbol X at position i of p is a nonterminal, then for every attribute a in to-be-bound-attrs(state), for every production p' in starts-by-binding(X, a), add item [state' = $[p', 1, \text{null}], k]$ to Agenda.

Completer: If state = $[\text{cat}, f, c]$ is inactive, then for every a-state = $[p, i, (d_1 \ldots d_n)]$ in union(S_k, init − states), if cat matches the rhs symbol at position i of p, the intersection of c and cover(a-state) is null, and k satisfies the expander constraint at position i of p, then add any item in advance(a-state, state, k) to Agenda.

Inverse-completer: If state $[p, i, (d_1 \ldots d_n)]$ is active, then for every i-state = $[cat, f, c']$ at S_k, if cat matches the symbol at position i of p, the intersection of cover(state) and c' is null, and k satisfies the expander constraint at position i of p, then add any item in advance(a-state, state, k) to Agenda.

- If there is an inactive state of the form $[X, f, u]$ in the parse table such that X = root-category of $\mathcal{G}$ and $u = C$, then succeed. Else fail.

6 Parse trace

The following trace of a parsing run uses the grammar from Example 4 together with the input shown in Figure 3. This trace picks the rectangle, indexed as 4, as the start element. Each step of the trace represents a state added to the table and includes the following information:

- <rule> or <category / feature vector>,
- <cover>, <indices>, and
- <source>.

Active and predictive states show the rule-name and dotted production in the form [<rule-name>: <dotted-rule>]. Inactive state categories and feature vectors are shown as #S(<cat> :<feat-1> <val-1> ... :<feat-n> <val-n>). The <cover> is shown as an integer representing a logical bit vector. For example, 15 represents the logical bit vector 1111, which in turn is the union of items indexed with integers 1, 2, 4, and 8. <Indices> is a list of input covers by which the inactive or active state is indexed in the parse table. The <source> information explains which subroutine produced the state.

1. [flowchart-2: Flow -> . P-block start end], 0, (4), init.
2. [flowchart-3: Flow -> . end P-block start], 0, (4), init.
3. [flowchart-1: Flow -> . start P-block end], 0, (4), init.
4. [conditional-1: P-block -> . decision P-block junction], 0, (4), init.
5. [conditional-3: P-block -> . junction P-block decision], 0, (4), init.
6. [conditional-2: P-block -> . P-block decision junction], 0, (4), init.
7. [basic-p-block: P-block -> . procedure], 0, (4), init.
8. #S(P-block :in 4 :out 4), 4, (4), scan 7.
9. [conditional-2: P-block -> P-block . decision junction], 4, (2), complete 6 with 8.
10. [conditional-2: P-block -> P-block decision . junction], 6, (8), scan 9.
11. #S(P-block :in 2 :out 8), 14, (8 2), scan 10.
12. [flowchart-2: Flow -> P-block . start end], 14, (1), complete 1 with 11.
13. [flowchart-2: Flow -> P-block start . end], 15, (16), scan 12.
14. #S(Flow :in 1 :out 16), 31, (16 1), scan 13.

7 Conclusion

The algorithm presented here is the first predictive, Earley-style algorithm that we know of for the full class of Atomic Relational Grammars. This class of grammars appears to be widely useful and is easily implemented through unification-based approaches; more specialized implementations closer in spirit to attribute grammars are also afforded. The primary problem addressed here is allowing for initialization at an arbitrary starting position in the input. The solution to this problem should carry over to other predictive parsers for multidimensional grammars as, for example, extended LR algorithms (see Costagliola et al., 1991).

Although the Earley-style algorithm presented here is of interest in its own right, there are remaining issues in exploring incremental, predictive parsing of visual language interfaces. To carry out the goal of providing an analogue of command completion in visual language interfaces requires at least two extensions beyond the work reported on here. First, more variations in scanning order are likely to be desired than what can be provided for here. Note that with the current algorithm, the ordering variants relative to a single global scanning order are restricted to local permutations within rules. What may be desired is the multidimensional analogue of predictive parsing of free-word-order languages that can scramble not only within grammatical constituents but also across constituents (subject to the connectness constraint). Further, algorithms more akin to island-based parsing (from a single island out) are likely to be preferable for interface parsing than the Earley-style algorithm presented here. Note that by following all permitted scanning orders reachable from a given start position, the Earley-style algorithm expands multiple islands in parallel, each of which may cover only part of the input globally processed so far.

Finally, a few short remarks on related literature. In the theoretical graph grammar literature, there have been recent results suggesting a natural class for a useful and general class of graph grammars, namely, context-free hypergraph grammars of bounded degree (Englefreit, 1992). An interesting line of research would be to investigate the relationship between Atomic Relational Grammars and Hypergraph grammars. The role of features and percolation in Atomic Relational Grammars seems to be quite similar to hyperedges and hyperedge replacement in Hypergraph Grammars. Elsewhere in the graph grammar literature, an active chart parsing algorithm for flowgraphs has been proposed (Lutz, 1989; Wills 1990) that is related to the parsing algorithm discussed here and in Wittenburg et al. (1991). Again, the exact relationship between flowgraphs and Atomic Relational Languages is worthy of investigation.

The most closely related parsing algorithms from the visual language literature are to be found in Tomita (1990) and Costagliola — Chang (1991), both of which extend Earley-style parsing into multidimensional domains. Although there is commonality at the level of parsing subroutines, indexing methods differ substantively. These differences arise in part because of different assumptions regarding the nature of the input and the allowable relations. Both these other proposals assume that the input is held in a grid of some kind with elements of equal size. Relations are defined accordingly. There are no such assumptions here.

Acknowledgements

This research was carried out in the Bellcore Computer Graphics and Interactive Media research group, directed by Jim Hollan. Thanks to Steve Abney, Bernard Lang, David Weir, and Louis Weitzman for discussions related to the topic of this paper.

References

Costagliola, G. — S.K. Chang (1991) "Parsing 2-D Languages with Positional Grammars." In: *Proceedings IWPT-91, Second International Workshop on Parsing Technologies*, 13-15 February 1991. 235–243. Pittsburgh, Pennsylvania: Carnegie Mellon University, School of Computer Science.

Costagliola, G. — M. Tomita — S.K. Chang (1991) "A Generalized Parser for 2-D Languages." In: *1991 IEEE Workshop on Visual Languages* (Kobe, Japan). 98–104.

Crimi, C. — A. Guercio — G. Nota — G. Pacini — G. Tortora — M. Tucci (1991) "Relation Grammars and their Application to Multidimensional Languages." In: *Journal of Visual Languages and Computing* 2(4), 333 – 346.

Earley, J. (1970) "An Efficient Context-Free Parsing Algorithm." In: *Communications of the ACM 13*, 94 – 102.

Engelfreit, J. (1992) "A Greibach Normal Form for Context-free Graph Grammars." In: Kuich, W. (Ed): *Automata, Languages and Programming: 19th International Colloquium*, Wien, Austria, July 1991, Lecture Notes on Computer Science 623, 138 – 149. Springer-Verlag.

Ferrucci, F. – G. Pacini – G. Tortora – M. Tucci – G. Vitiello (1991) "Efficient Parsing of Multidimensional Structures." In: *1991 IEEE Workshop on Visual Languages* (Kobe, Japan). 104 – 110.

Flasinski, M. (1988) "Parsing of edNLC-Graph Grammars for Scene Analysis." In: *Pattern Recognition* 21, 623 – 629.

Flasinski, M. (1989) "Characteristics of edNLC-Graph Grammar for Syntactic Pattern Recognition." In: *Computer Vision, Graphics, and Image Processing* 47, 1 – 21.

Golin, E. J. (1991) "Parsing Visual Languages with Picture Layout Grammars." In: *Journal of Visual Languages and Computing* 2, 371 – 393.

Golin, E.J. — S.P. Reiss (1990) "The Specification of Visual Language Syntax." In: *Journal of Visual Languages and Computing* 1, 141 – 157.

Lutz, R. (1989) "Chart Parsing of Flowgraphs." In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 116 – 121.

Helm, R. — K. Marriott (1986) "Declarative Graphics." In: *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science 225, 513 – 527. Springer-Verlag.

Helm, R. — K. Marriott (1990) "A Declarative Specification and Semantics for Visual Languages." In: *Journal of Visual Languages and Computing* 2(4), 311 – 331.

Tomita, M. (1990) "The Generalized LR Parser/Compiler V8-4: A Software Package for Practical NL Projects." In: *Proceedings of COLING-90*, Volume 1, 59 – 63.

Tomita, M. (1991) "Parsing 2-Dimensional Language." In: Tomita, M. (Ed): *Current Issues in Parsing Technology*, 277 – 289. Kluwer Academic.

Weitzman, L. — K. Wittenburg (1993) "Relational Grammars for Interactive Design." In: *Proceedings of 1993 IEEE/CS Symposium on Visual Languages*, August 24-27, Bergen, Norway.

Wills, L. (1990) "Automated Program Recognition: A Feasibility Demonstration." In: *Artificial Intelligence* 45, 113 – 171.

Wittenburg, K. — L. Weitzman — J. Talley (1991) "Unification-Based Grammars and Tabular Parsing for Graphical Languages." In: *Journal of Visual Languages and Computing* 2(4), 347 – 370.

Wittenburg, K. (1992a) "Earley-style Parsing for Relational Grammars." In: *Proceedings of the 1992 IEEE Workshop on Visual Languages*, Seattle, Washington, Sept 15-18, 1992. 192 – 199.

Wittenburg, K. (1992b) *The Relational Language System*. Bellcore Technical Memorandum TM-ARH-022353.

Wittenburg, K. (1993) "F-PATR: Functional Constraints for Unification-based Grammars." In: *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, 216 – 223.

APPENDIX

Probabilistic Incremental Parsing in Systemic Functional Grammar

A. Ruvan Weerasinghe*
Robin P. Fawcett
Computational Linguistics Unit, University of Wales College of Cardiff, UK.

July 5, 1993

Abstract

In this paper we suggest that a key feature to look for in a successful parser is its ability to lend itself naturally to semantic interpretation. We therefore argue in favour of a parser based on a semantically oriented model of grammar, demonstrating some of the benefits that such a model offers to the parsing process. In particular we adopt a systemic functional syntax as the basis for implementing a chart based probabilistic incremental parser for a non-trivial subset of English.

1 Introduction

The majority of the research in the field of Natural Language Understanding (NLU) is based on models of grammar which make a clear distinction between the levels of syntax and semantics. Such models tend to be strongly influenced by formal linguistics in the general Chomskyan paradigm, and/or by mathematical formal language theory, both of which make them conducive to computer implementation. Essentially, these models constitute an attempt to 'stretch' techniques that have been applied successfully to parsing artificial (and so unambiguous) languages, in order to apply them to natural language (NL).

In recent years, however, models of language that are derived from the text-descriptive tradition in linguistics have emerged as potentially relevant to NLU. These models emphasize the *semantic* and *functional* richness of language rather than its more formal and syntactic properties. Such models may challenge widely held assumptions, e.g. that the key notion in modelling syntax is grammaticality, and that this is to be modelled using some version of the concept of a phrase structure grammar (PSG)[1].Since such grammars emerge from use in analysing texts, they have something in common with the sort of grammars that tend to be used in corpus linguistics. To date the strongest influence of these grammars has been in Natural Language Generation (NLG) (Fawcett et al., 1993; Matthiessen, 1991).

The semantic orientation of functional grammars, however, is to some extent in conflict with the better understood techniques for parsing syntax. The research described in

*On leave from the Department of Statistics and Computer Science, University of Colombo, Sri Lanka.

[1]It is evident, however, that researchers working in the formal linguistics paradigm have in recent years increasingly realized the importance of the functional aspects of language, e.g. in augmenting their models with syntactico-semantic features.

this paper presents a probabilistic approach to parsing that yields a *rich syntax*, using a systemic functional grammar (SFG).In doing so, however, it also shows how some of the techniques used in traditional syntax parsing can be adapted to serve as useful tools for the problem. It will be shown that our parser is able to produce richly annotated ***flat*** parse trees that are particularly well-suited to higher level processing.

The main contributions to the formal specification of SFG, as they affect NLU, have been by Patten and Ritchie (1986), Mellish (1988), Patten (1988), Kasper (1987) and Brew (1991). These have mainly been concerned with the reverse traversal of system networks in order to get at the features from the items (words). They all conclude that systemic classification is NP-hard, but seek to isolate tractable sub-networks in order to be able to optimise reversal. It is thus apparent that a direct reverse traversal of the networks may not be the best approach to parsing in SFG.

Work of a more implementational nature is reported in Kasper (1988), Atwell et al. (1988) and Matthiessen (1991). The common approach to parsing systemic grammar in these has been to employ a 'cover grammar' for preprocessing the syntactic structure of the input string (instead of attempting to directly reverse the networks and realization rules), and then, as a second stage, to do the semantic interpretation by accessing the features contained in the system networks. O'Donoghue (1991b) suggests one possible way to avoiding this, namely by the use of a 'vertical strips parser'. This extracts the 'syntax rules' that are implicit in the grammar through analysing a corpus of text randomly generated by the generator (GENESYS[2]). His approach has the advantage that it addresses the problems of maintainability and consistency of the grammar (as used by both the generator and the parser), but it runs into problems of search space, and suffers from the limitation that the information is extracted from *artificial* random generation.

The current parser is an attempt to overcome the latter problem - but not at the expense of the former. The major emphases of the parser therefore can be stated as follows:

1. To maintain a close correspondence between the syntactic representation and the semantic representation which is to be extracted from it (this having implications for possible interleaved processing).

2. To obtain a syntactically and functionally rich parse tree (even when there is some ungrammaticality in the input).

3. To improve efficiency by (a) parsing incrementally and (b) guiding the parsing process by probability based prediction and the use of feature unification.

To this end we reject the strategy of adopting a PSG-type 'cover grammar', in the style of Kasper (1988) and adopt instead a systemic syntax as the basis of the parser. This is stored in the form of

1. Componence, filling and exponence tables, as described in section 2.3 and

2. The transition probabilities of the elements in the componence tables

The output of the parser's incremental evaluation of the parse can be exploited by a semantic interpreter of the kind described by O'Donoghue (1991a, 1993); see also Fawcett

[2]GENESYS is the main generator in the COMMUNAL project; It stands for GENErate SYStemically ;COMMUNAL stands for the Convivial Man Machine ... Using NAtural Language, and is a DRA sponsored project at the University of Wales College of Cardiff, UK.

(1993). Essentially, this runs the system networks in reverse to collect the features required[3].

In the rest of this paper we will introduce the concept of 'rich syntax' with respect to SFG (section 2), and then describe the techniques we adopt for parsing it (section 3). Finally, in section refconclusions we will evaluate the work done so far and discuss its limitations and future work envisaged.

2 Parsing for rich syntax

2.1 Systemic Functional Grammar

Before we describe the nature of systemic functional syntax, we need to point out that the syntactic structures (to be discussed in section 2.2) are not the heart of the grammar, but the outputs from the operation of the SYSTEM NETWORKS and their associated REALIZATION RULES[4].

SFG is a model of grammar developed from a functional view of language which has its roots in the work of Firth and the Prague School; Its major architect is Halliday. The more well known computer implementations of it have been developed mainly in the complementary field of Natural Language Generation (NLG). Some of these include Davy's PROTEUS(1978), Mann and Matthiessen's NIGEL(1985) and Fawcett and Tucker's GENESYS(1990). One significant NLU system based upon systemic syntax is Winograd's SHRDLU(1972).

The core of the grammar consists of a great many choice points, known as *systems*[5], at each which the system must take one path or another by choosing one of two (and sometimes more) semantic features. Quite large numbers of such systems combine, using a small set of AND and OR operators, to form a large network, as shown in figure 1. The big lexicogrammar which the parser described here is designed to work with has about 600 grammatically realized systems. The network is traversed from left to right, and each such traversal generates a 'selection expression' (i.e. a bundle) of features. Some of these have attached to them REALIZATION RULES, and it is these which, one by one, combine to build the semantically motivated 'syntax' structures that we shall describe in section 2.2.

For example, consider the fragment of a network shown in figure 1 below[6]. It shows a simplified version of the current network in the 'midi' version of the COMMUNAL grammar. What is not shown here is a detailed formal specification of the realization rules for the features collected by following the various possible pathways through the network. The first two realizations are however expressed informally: i.e. the meaning of [information] plus [giver] is realized by having the Subject (S) before the Operator (O), if there is an Operator, and if not by having the Subject before the Main Verb (M).

It should be noted that in the 'full' grammar there are probabilities attached to each system (or choice point). This enables the model to escape from the conceptual prison of the concept of grammaticality and enables us to account for very unlikely, yet possible choices being made.

[3] An alternative would be to have a separate compositional semantics component based on the functional paradigm described in this paper.

[4] Readers familiar with how a systemic functional grammar works may wish to skip this section.

[5] For these, and for an overview of the role of SFG in NLG, see Fawcett et al. (1993).

[6] For those familiar with SFL, there may be some interest in comparing the network given here with the traditional network for MOOD. It has been made much more explicitly semantic than the traditional MOOD network (which begins with [indicative] or [imperative], and then, if [indicative], either [declarative] or [interrogative]).

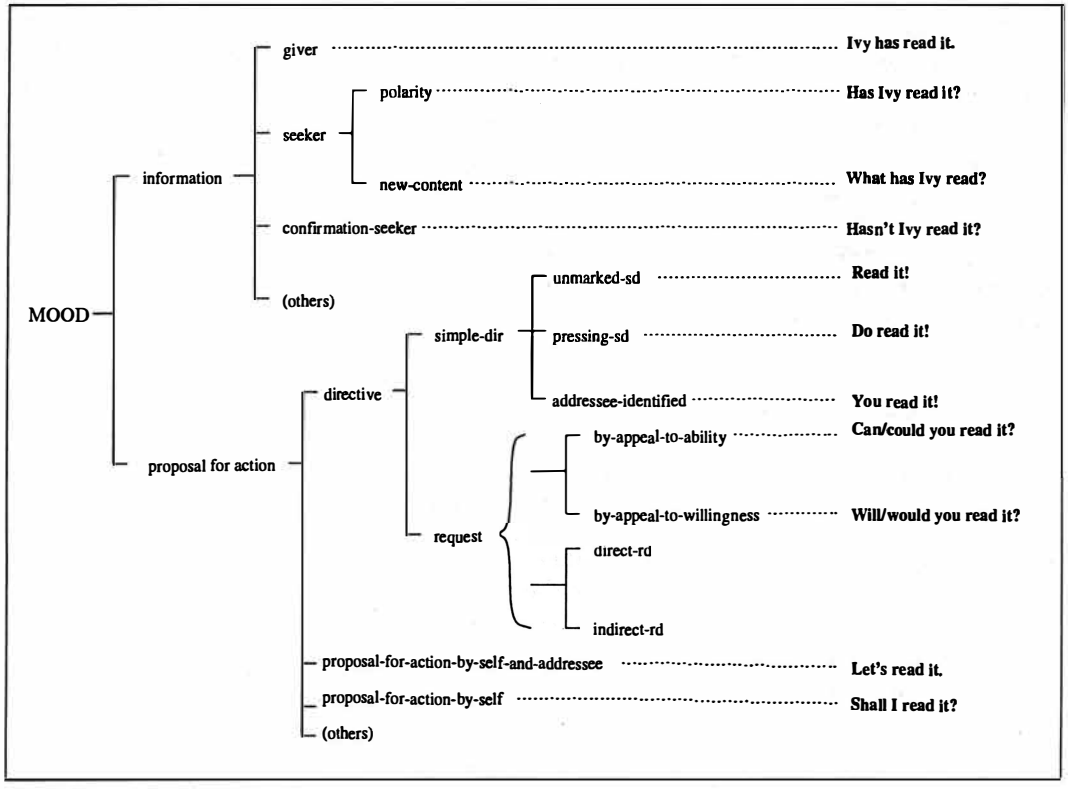


Figure 1: A simplified system network showing some of the meaning potential for MOOD in English (excluding much, e.g. POLARITY and realizations in tags and intonation)

2.2 Semantically motivated systemic functional aspects of the model

The syntactico-semantic analysis produced by the parser differs from traditional syntactic parse trees in at least the following four important ways.

1. Firstly, our model of 'syntax' distinguishes between the relationships of :

   (a) COMPONENCE, whereby a particular UNIT such as a nominal group (a 'full' noun phrase; denoted by 'ngp' in our systemic syntax) is composed of ELEMENTS (functional categories) *and*

   (b) FILLING, whereby such a UNIT 'fulfills', as it were, the functional role of the element it fills.

So, for instance, a ngp can have (among others) the components deictic determiner (dd), modifier (m) and head (h). At the same time it will 'fill' either the element functioning as Subject (S), a Complement (C1/2), a 'Completive' of a prepositional group, or some other element.

Together, the two concepts of filling and componence provide for (two types of) recursion in this model. Firstly, for example, a modifier in a nominal group can itself be filled by, among other things, a nominal group; this is the SFG approach to the phenomenon known (misleadingly) sometimes as 'noun-noun compounding', e.g. as in *their luxury flat sale*, where it is the ngp *luxury flat* (and not just the noun *flat*) that modifies *sale*.

The second type of recursion that is accounted for is COORDINATION. In cases such as *my brother and his wife (have arrived)*, both nominal groups (*my brother* and *and his wife*) fill the role of Agent (which is conflated with the Subject; see (c) below); they are jointly the Agent in the Process of *arriving*. Without the concept of filling, we would be forced to represent this relationship as if it were componence.

2. Secondly, the emphasis on *function* in the model is evident in the elements which constitute the final non-terminals in the syntax tree, which are categorised in terms of their *function* in the unit above, rather than as 'word classes'. In this scheme, the term 'noun' for example is a label for *one* of the classes of words which may expound the head of a ngp. Others may include pronouns, proper names or *one(s)*.

   Again, *very* is not treated simply as another 'adverb' (which misleadingly suggests that it functions similarly to *quickly*, etc), but as a 'temperer'. This is because it typically 'tempers' a quality of either a 'thing', as in (1b) below, or a 'situation', as in (1c). It is thus an element of what is here termed a 'quantity-quality group', in which the 'head' element, which expresses the 'quality', is termed the 'apex' (a) and the 'modifier' element, which 'tempers' it by expressing a 'quantity' of that quality, is termed a 'temperer'. This functional enrichment of the syntax provides a natural way to account for the difference between the functions of *very* and *big* in sentences 1a and 1b.

(1a) She noticed the big fat man.

(1b) She noticed the very fat man.

(1c) She ran very quickly to the window.

Finally, also note that, the grammar allows some (but not all) elements to be either 'expounded' by lexical or grammatical items or 'filled' by a syntactic unit. For example, consider the quantifying determiner (dq), which is EXPOUNDED in (2) and (3) by the ITEMS *a* and *one*, and FILLED by the nominal group (UNIT) *a bag* (dq h) in (4).

(2) He was a very interesting man.

(3) I'd like one cabbage, please.

(4) I'd like a bag of potatoes please.

Points 1 and 2 above, reflect and respect a specific commitment to maintaining the closest possible correlation between the units recognized in syntax and those recognized in semantics. Thus, in the unmarked case, a CLAUSE (cl) realizes a SITUATION (= roughly 'proposition') and a NOMINAL GROUP (ngp) realizes a THING ('object'). Hence our parser produces broad flat trees rather than those with multiple (often binary) branching; the 'work done' in models of the latter sort by the branching is done in our model by richer labelling. ie. Richer labelling reduces the need to represent relations by

distinctive tree structure variations, thus enabling us to restrict the branching to the definition of units that are semantically motivated. And this in turn greatly facilitates the transfer of the output from the parse to the next stage of processing.

3. Third, we also consider that the notion of ***absolute grammaticality***, which is intrinsic to phrase structure type grammars to be fundamentally misleading. Instead, we take a probabilistic approach to the question of what element may (or is unlikely to) follow what other element in a unit. The concept of ungrammaticality is thus simply one end of a continuum of probabilities from 0% to 100%. In this respect, our parser has characteristics in common with stochastic approaches to parsing, and so embodies, in effect, a *hybrid* model.

   Hence our parser accommodates some measure of ***ungrammaticality*** in the input text, and tries to extract ***whatever*** functional information it can from it – rather than rejecting it.Consider the example sentence below.

   (5) Car sales, in spite of the recession, *was* up by more than five per cent.

   The type of unification parser which enforces subject-verb agreement would simply return the verdict 'ungrammatical' on encountering such an utterance. Chart based parsers are a slight improvement, in that they would output the 'analysable' fragments of the sentence. Because our parser's goal is to return *some* semantically plausible interpretation, it returns a well formed parse tree out of such input[7].

4. Finally we should point out that our parser is helped to build its functionally rich output using the familiar concept of feature unification. Many other modern parsers use feature unification as a means of constraining the ever-growing parse forest caused by local ambiguity, while then passing up the 'unified features' for higher semantic processing. In a functionally oriented grammar such as SFG (in which the system networks from which the structures are generated are themselves entirely made up of semantic features) it is particularly natural to supplement the syntax tree through such 'percolated' features[8].

   Thus for instance, the features [manner], [place] and [time] are 'percolated' up from the items ***unexpectedly***, ***to Cardiff*** and ***on Friday*** respectively in (6).

   (6) He unexpectedly came to Cardiff on Friday.

2.3 The syntactic coverage of the parser

As will be evident from what has been said so far, there is no set of PSG-type re-write rules that specifies the syntax. Instead there are semantic features whose realization rules, collectively, build up syntactic structure. The information that a parser needs to have available to it is only implicitly present in the generator, and it is not in a form that is readily usable by the parser. O'Donoghue (1991b) explores one possible way of overcoming this problem, namely by extracting the 'rules' (or 'legal sequences') from sentences randomly generated by the generator (GENESYS). Our approach

[7] We take the view that such grammatical features are in fact not normally of any great help in disambiguation, and hence not of much use in further processing.

[8] Note that the use of features for constraining the parse forrest using for instance number agreement is not done here.

is to extract from the system networks and realization rules the information about syntax that is relevant to the work of the parser, and to state it in a form that is more amenable to this task[9].

The four major types of units recognized by the parser's syntax are the clause (cl), the nominal group (ngp), the prepositional group (pgp) and the quantity-quality group (qqgp)[10].

Of these,four units, the clause has by far the most complex and variable syntax. The elements of the ngp, pgp and qqgp on the other hand can be considered for practical purposes to be fixed, and the presence or absence of elements within such groups is reflected in our model in the transition probabilities (see section 3.1). Because of the fixed sequence of elements in these groups, we can at this point use a re-write rule notation to represent these.

ngp → dq, vq, dd, m, mth, h, qth, qsit
pgp → p, cv
qqgp → t, a, f [11]

Here, the '→' is used to denote the COMPONENCE relationship. Thus, for instance, a pgp can be composed of a preposition (p) and a completive (cv).

However, the above specifications have a number of grave limitations. They fail to show (1) those elements that are optional, (2) the degree of optionality, and (3) the dependencies that may hold between them, absolutely and relatively (e.g. there can be no vq if there is no dq, and it is highly unlikely that there will be a dq without an h). As we shall see, it is the information about probabilities that captures these facts.

The most complex of the groups, the clause, has a more variable potential structure which here we denote (for convenience) by the re-write rule[12]:

cl → B, A*, C2, O# ,S, O#, N, I, A*, X*, M, Cm, C1, C2, Cm, A*

As we shall shortly see, the information about adjacent elements expressed in these specifications, together with other vital information on optionality and probabilities, is made available to the parser in a somewhat different form.

A second type of information required by the parser is a set of statements about FILLING, i.e. about the elements which units can fill, thus [13] :

cl : A, C2, f
ngp : A, C1, C2, cv, mth, dq
pgp : A, C1, C2, qth, f
qqgp : m, A, C2, dq

Here, the ':' stands for the filling relationship. So, for example, the clause (cl) can fill an Adjunct, a second complement (C2) or a finisher (f).

To illustrate the type of syntax tree these two relationships together provide, consider

[9] We hope to be able to device a technique for automatic extraction of 'rules' from the system networks in future versions of the parser, but we defer this task for the present as it has been shown to be possible (O'Donoghue, 1991b).

[10] We should state here that the syntax described below handles only a subset of the 'midi grammar' contained in the system networks of GENESYS referred to above, and that we have nothing to say here about phenomena such as 'raising' and 'long-distance dependency' (though many aspects of discontinuity are already covered within GENESYS, and these types of phenomena are now being considered in the SFG framework).

[11] The key to the list of elements used in the parser is given in the Appendix.

[12] Here, * denotes recursive elements while # denotes that the Operator element can be 'conflated' with the functions of X (auxiliary) or M (main verb).

[13] Note that B, O, N, I, X and Cm are directly expounded by items.

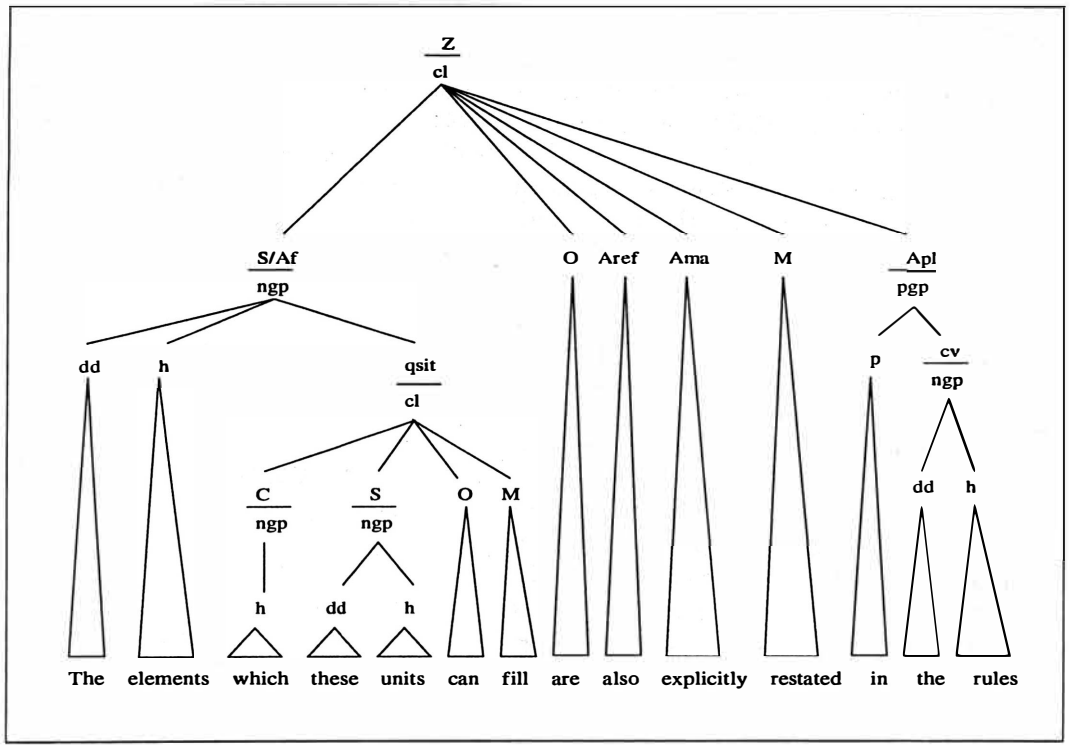


Figure 2: A Systemic Functional Analysis of a sentence

the typical SFG analysis of a sentence (Z) shown in figure 2 [14].

Note that in this analysis, the fragment *the elements which these units can fill* both corresponds to a single nominal group (filling the element of Subject and the participant role of Affected) and constitutes a single semantic 'thing' (or 'object').

We would argue, with Winograd (e.g. Winograd, 1972), that such 'flat' tree representations lend themselves more naturally to higher level processing than do trees with many branchings, because each layer of structure corresponds to a a semantic unit, and ultimately to a unit in the 'belief' representation.

[14] See appendix for 'conflation' abbreviations.

There is no genuine equivalent relationship to this in a PSG, because such grammars do not have the 'double labelling' of nodes in the tree as both element and unit (or, with coordination, units) described above. That is, there is no distinction between componence (unit down to element) and filling (element down to unit(s)). From the viewpoint of a parser, the relationship we are considering here is a unit-up-to-element table. Here the probabilistic information is extremely valuable; it is useful for the parser to know, for example, that it is relatively unusual for a clause to fill a Subject, but that a clause fairly frequently fills a Complement or Adjunct.

We have been considering the 'unit up to

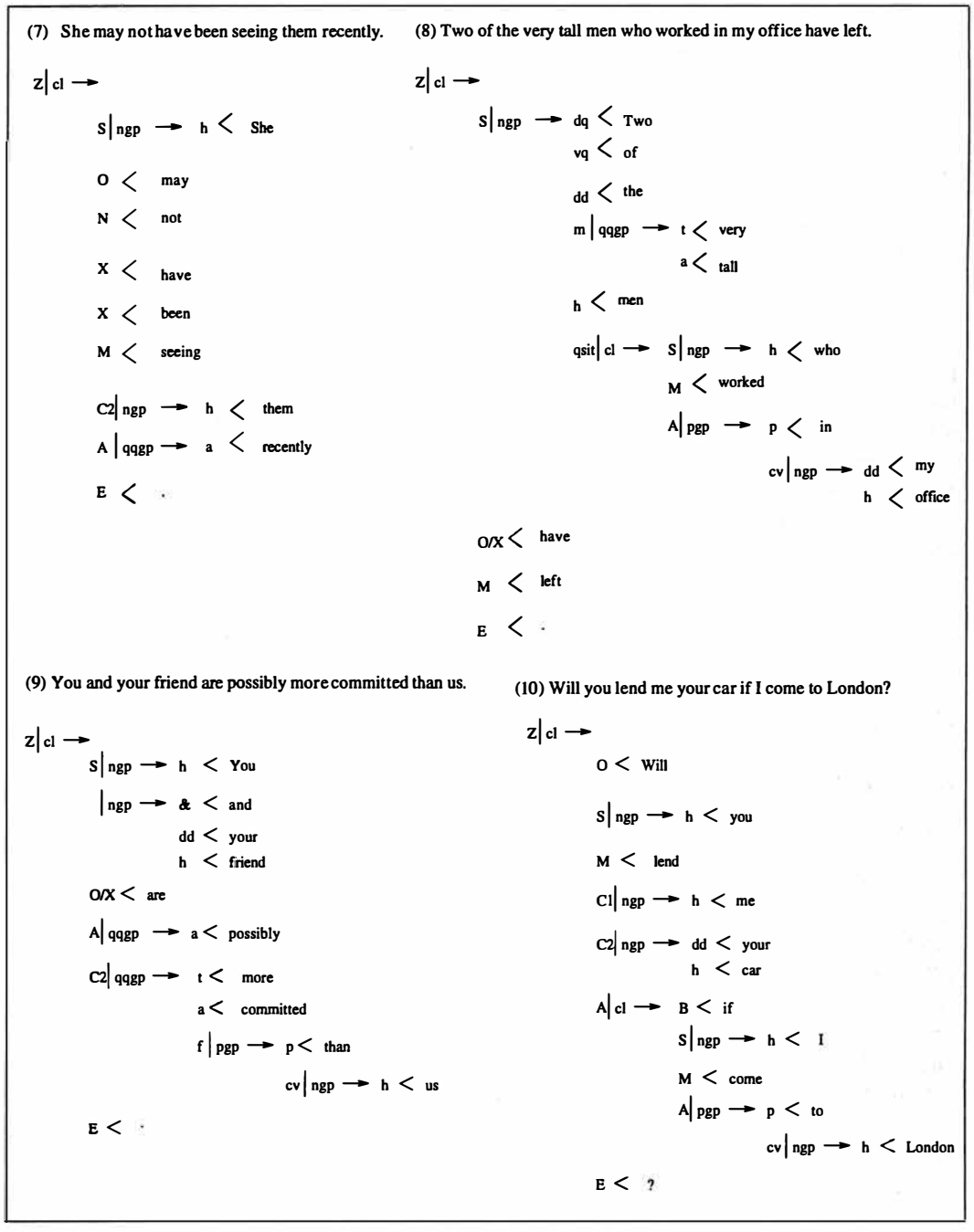


Figure 3: Some examples of the output of the parser

element' relationship of componence. Finally, there is the similar 'item up to element' relationship of EXPONENCE. This is a list of all the items (roughly, 'words') to be accepted by the parser , with the probabilities - for each sense of each word - of what elements each may expound (placed in order). The difference from the previous information source is that it is a very large and constantly modifiable component; the coverage of the unit up to element tables is in comparison quite limited (and less liable to revision in the light of successfully parsed new texts). This third component is therefore the equivalent in our parser of what is often termed the 'lexicon'.

As is shown by the example in figure 2, the SFG approach makes possible the output of syntactically rich, semantically oriented 'flat' tree parse structures. The typical outputs from the parser shown in figure 3 should, it is hoped, give a picture of the kind of data covered by the syntax, and so by the parser[15].

3 How the parser works

3.1 The basic algorithm

The fundamental strategy adopted in parsing for the rich functional syntax described in sections 2.2 and 2.3, is an adapted form of bottom up chart parsing with limited top down prediction. One of the main reasons for the adaptation of the chart parsing algorithm is to account for some of the context sensitivity exhibited by the SF syntax. For example, the possibility of an 'Operator' occurring after a Subject is dependent on it's non-occurrence before it. Similarly, certain types of Adjunct are mutually exclusive within a given clause. For this reason, our parser has lists of 'potential structure' templates (as shown in simplified form in section 2.3) instead of the usual CF-PSG type rules. These are augmented by the element transition probability tables and a probabilistic lexicon, to assist the adapted probabilistic chart parser implemented here.

Hence, the chart is composed of edges, each with a list of the elements that can 'potentially' occur following it, together with optionality and mutual exclusivity constraints, features associated with the edge and a unique probability score representing its likelihood of occurrence. As in the case of standard 'active' chart parsers, 'active' or hypothesis bearing edges too are maintained in a similar way.

The unification of edges is used only to perform a 'percollatory' function rather than a 'restrictive' one, so as to give less importance to the concept of 'grammaticality'. The aim of this is to allow some 'ungrammaticality' in order to extract at least *some* meaning from *any* utterance.

The probabilities themselves are calculated from three sources:

1. The item probabilities contained in the exponence table (the 'lexicon').
2. The filling probabilities for each unit.
3. The transition probabilities between elements within a unit.

In a given application of the 'fundamental rule', three component probabilities are used in working out a weighted geometric mean. It is our observation that, as Magerman and Marcus (1991) point out, joint probabilities calculated as products are not accurate estimates of such likelihoods owing to the events considered violating the independence assumption. The three probabilities thus affecting the new edge created are :

1. The probability of the 'active' edge in the 'attachment'.

[15]Note that at this stage the parser does not yet assign participant roles.

2. The probability of the 'inactive' edge of the 'attachment' .
3. The probability of the transition of elements involved in the 'attachment'.

For example, consider the fragment *the man* shown in figure 4.

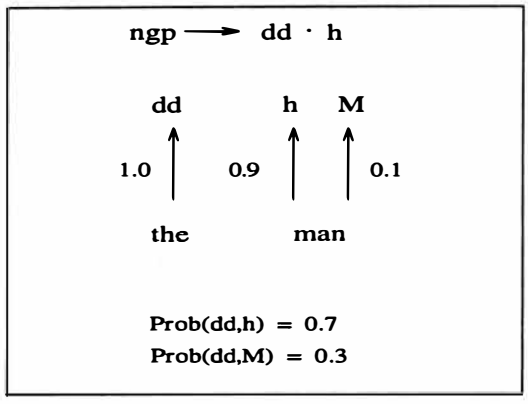


Figure 4: Edge creation with probabilities

In this situation, the two edges *the* and *man* would invoke the hypothesis (using the usual 'chart' notation):

ngp → dd . h

where the ngp is 'looking for' a head. In the ensuing unification of this active edge with h(man), we consider the probabilities of :

1. the active edge ngp(dd(the))
2. the inactive edge h(man) and
3. the transition (dd,h).

A geometric mean of the probabilities of 1 and 2 and a weighted 3 is attached to the new (inactive) edge ngp(dd(the), h(man)), that is thus formed. The weighting on the third component makes it favour the transition predictions over those of filling and exponence.

Subsequently, the filling probabilities of S, C, cv etc. will be considered. In the case where the above fragment begins the input string, the clause level transition probabilities will heavily favour the S to be the element being filled by the ngp (with a high score for the transition (\$,S)) than C or cv.

Consider the following example sentence to see how such a probabilistic model can assist in arriving at a correct analysis of a clause with lexical ambiguity :

(11) Did you notice him?

In this example, though *notice* could be both a head (noun) or a verb, the transition probability of head-head is very low. (Noun-noun compounding will not score well as the probability of *you* being able to fill a modifier is negligible). On the other hand, the transition probability of S-M is very high and so *notice* will be parsed as a M in the leading 'theories'.

Finally, consider the following 'garden path' type sentence to see how our probabilistic model copes with this type of ambiguity:

(12) The cast iron their clothes.

According to the COBUILD dictionary, *cast* is most commonly a noun(h) or a verb(M) while *iron* is most commonly a noun(h), but could also be a verb(M) and more rarely an adjective(a). A part-of-speech tagger encountering this input string will need to determine which of the transitions (dd,h,h), (dd,h,a), (dd,h,M), (dd,M,h) etc. are more likely. A tagger based on lexical co-occurrences or part-of-speech might well favour (dd,h,h), the former as it could have information about *cast* and *iron* being able to follow each other in this way and the latter to account for noun-noun compounding.

Our parser on the other hand, though initially favouring this theory like the class based tagger, will also advance the theory containing *iron* as a main verb(M). Once a certain

'height' of the parse tree is reached however, the probability score of theories treating *iron* as a noun(h) will diminish while those treating it as a verb(M) will be re-inforced by the high transition probabilities of the higher elements (S,M) and (M,C).

It is the availability of these 'higher' functional syntax level transition probabilities to the parser, that we suspect will enable our parser to perform better than (conventional) pure probabilistic part-of-speech level models.

3.2 The interactive interface

A major secondary goal of our work is to build a parser which can function as the front end to a complete interactive NLP system (COMMUNAL). To this end we have developed an interactive interface to the parser. Incoming items are tagged to focus the search space using a character reading input routine, which is responsible for providing (incrementally) the parser with a 'clean' input by :

1. Tagging punctuation according to the elements they expound.
2. Handling the syntax of large and decimal numbers.
3. Flagging abbreviations appropriately.
4. Signalling unknown words or assigning likely elements which might expound them

The 'final non-terminals' output by this routine are input to the parser incrementally, while simultaneously accepting further input. Thus by the time the user input is completed (by the tagger encountering an 'Ender' item) much of it has already been analysed by the parser.

The incremental nature of processing at this syntax level can be further exploited at higher 'interpretive' stages of analysis because of the well annotated 'flat' parse trees produced and their (near) one-to-one correspondence with semantic objects in the SFG adopted. (See Van der Linden (1992, p. 225) for reasons why traditional PSG-type grammars cannot in general be parsed incrementally in this way).

As an example, consider the following sentence.

(13) The boy with long hair saw Jill in the park.

Here, as soon as the user starts to input *Jill*, the item *saw* is tagged, with its syntactic context guiding the decision. Meanwhile, *The boy with the long hair* has already been identified as a nominal group (unit) with certain (quite limited) semantic properties, and it is thus ready for verification as, say, ⟨person102⟩ very early in the parse process.

4 Conclusions

4.1 Evaluation as a general functional parser

It is necessary firstly to evaluate our parser with respect to the richly annotated functional parse it produces. While time and space efficiency issues of the algorithm have not been brought to bear too heavily on the work done, the techniques adopted are general enough to be used for parsing other functional grammars represented as 'structural templates' (and supplemented by features and transition probabilities, and filling and exponence tables), with minimum modification to the algorithm itself.

The information contained in the flat parse trees constructed by the parser, while being richer in content, also allow for natural interleaving of syntax with higher semantic and pragmatic processing.

In this sense, we consider the current parser to be a successful precursor to a fully proba-

bilistic chart parser for functionally rich grammars.

More detailed formal evaluation, both of time-space efficiency of the algorithm and the parser's accuracy in analysing free text needs to be postponed for the present, until the parser is 'trained' on the fully systemically (hand) parsed Polytechnic of Wales (POW) corpus[16]. At the time of writing, a tool for the extraction of the necessary probabilities has been implemented (Day, 1993), though it needs as yet to be linked to the parser's probability module.

4.2 Evaluation as a front-end to COMMUNAL

Though the general algorithm is concerned with text parsing, our specific area of application is to use the parser as a front-end to the COMMUNAL NLP system, which is already equipped with a large systemic functional grammar embodied in its generator GENESYS. For this reason, the parser is equipped with an interactive interface which acts on input in an incremental way. It is also able to achieve a significant coverage of the 'midi' version of the GENESYS grammar. Our thesis is that this prototype parser will lend itself to being substantially extended to cover other complex grammatical phenomena handled by the 'full' version of the grammar, without the need to make any major alterations to the techniques employed in it.

4.3 Limitations and further work

One of the main limitations of the integrity of the system is that of the parser needing to be manually supplied with grammatical information embodied within the generator, GENESYS. An urgent need therefore is for a technique for extracting this information directly without human intervention. This would enable any grammar represented in system network notation to be compiled into a parsable form.

The main source of lexical probabilities for the parser has been West (1965), while element transition probabilities have been extracted (using the aforementioned interactive tool) from the POW corpus. For a more consistent approach non-reliant on human intervention, more work is needed on developing a non-interactive version of the parser which is able to train on hand parsed corpora.

The improvement of these aspects of the system will allow the current parser to be used as a robust 'real text' parser and to be incorporated into a NLU system capable of true interleaved processing.

[16]This is available from ICAME's archive at the Norwegian Computing Centre for the Humanities in Bergen, Norway.

Reference

Atwell, E. S., Souter, C. D., & O'Donoghue, T. F. (1988). Prototype parser 1. Tech. rep. 17, Computational Linguistics Unit, University of Wales College of Cardiff, UK.

Brew, C. (1991). Systemic classification and its efficiency. *Computational Linguistics*, *17*(4).

Davy, A. (1978). *Discourse production: A computer model of some aspects of a speaker*. Edinburgh University Press, Edinburgh, UK.

Day, M. D. (1993). The interactive corpus query facility: a tool for exploiting parsed natural language corpora. Master's thesis, University of Wales College of Cardiff, UK.

Fawcett, R. P. (1993). A generationist approach to grammar reversibility in natural language processing. In Strzalkowski, T. (Ed.), *Reversible Grammar in Natural Language Generation.* Dordrecht: Kluwer.

Fawcett, R. P., & Tucker, G. H. (1990). Demonstration of genesys: a very large, semantically based systemic functional grammar. In *The 13th International Conference on Computational Linguistics (COLING-90)*, pp. 47–49.

Fawcett, R. P., Tucker, G. H., & Lin, Y. Q. (1993). How a systemic functional grammar works: the role of realization in realization. In Horacek, H., & Zock, M. (Eds.), *New Concepts in Natural Language Generation: Planning, Realization and Systems.* Pinter, London.

Kasper, R. T. (1987). A unification method for disjunctive feature descriptions. In *Proceedings of the 25th Annual Meeting of the Association of Computational Linguistics.*

Kasper, R. T. (1988). An experimental parser for systemic grammars. In *The 12th International Conference on Computational Linguistics (COLING-88).*

Magerman, D. M., & Marcus, M. P. (1991). Pearl: A probabilistic chart parser. In *Procedings of the 2nd International Workshop on Parsing Technologies.*

Mann, W. C., & Matthiessen, C. (1985). Nigel: A systemic grammar for text generation. In Freedle, R. O. (Ed.), *Systemic Perspectives on Discourse.* Ablex.

Matthiessen, C. M. I. M. (1991). *Test generation and systemic functional linguistics.* Pinter, London.

Mellish, C. S. (1988). Implementing systemic classification by unification. *Computational Linguistics, 14*(1).

O'Donoghue, T. F. (1991a). A semantic interpreter for systemic grammars. In *Proceedings of the ACL Workshop on Reversible Grammars.*

O'Donoghue, T. F. (1991b). The vertical strip parser : a lazy approach to parsing. Report 91.15, School of Computer Studies, University of Leeds, UK.

O'Donoghue, T. F. (1993). Semantic interpretation in a systemic grammar. In Strzalkowski, T. (Ed.), *Reversible Grammar in Natural Language Generation.* Dordrecht: Kluwer.

Patten, T. (1988). *Systemic Text Generation as Problem Solving.* Cambridge University Press.

Patten, T., & Ritchie, G. (1986). A formal model of systemic grammar. Research paper 290, Deptartment of AI, Edinbutgh University, UK.

Van der Linden, E.-J. (1992). Incremental processing and the hierarchical lexicon. *Computational Linguistics, 18*(2).

West, M. (1965). *A general service list of english words.* Longmans.

Winograd, T. (1972). *Understanding Natural Language.* Academic Press Inc.

Appendix

Symbol	Name	Also Known As	Function
Clause elements:			
cl	Clause	Sentence	Realizes a 'situation'
&	Linker	Conjunct(coord)	Links two 'equal' units
B	Binder	Conjunct(subord)	Binds subordinate unit into a higher unit
A	Adjunct	Adverbial/Prepositional phrase	Realizes circumstantial roles, etc. in the clause
C1/2	Complements	Object(direct/indirect)	Realizes main participant roles in the clause (together with S)
O	Operator	First Auxiliary	Realizes mood, negation, emphasis or polarity, tense
X	Auxiliary	Auxiliary	Realizes tense, aspect, passives
O/M	Operator/Main Verb	Modal Verb	Operator functioning as the Main Verb of the clause
S	Subject	First NP	Specifies the subject role of the clause
N	Negator	Negator	Negates clause
I	Infinitive	Infinitive	Used in infinitive clauses
M	Main Verb	Verb	Specifies the process, constrains the roles in clause and tense
Cm	Main-Verb-completing complement	Particle	Completes the meaning of the Main Verb
Nominal group elements:			
ngp	Nominal group	Noun phrase	Realizes 'things'
dq	Quantifying determiner	Determiner	Quantifies the nominal group
vq	'of' element	Preposition	Shows 'selection' relationship
dd	Deictic determiner	Determiner	Marks definiteness in the nominal group
m	Modifier	Adjective	Modifies the 'head' of the group

mth	Thing modifier	Noun modifier	Modifies the 'head' of the group
h	Head	Noun	Marks the head noun or is a pronoun or proper name
qth	Qualifier	PP	Modifies the 'head' by a prepositional group
qsit	Qualifier	Relative clause	Modifies the 'head' by a clause

Prepositional group elements:

pgp	Prepositional group	PP	Realizes a 'minimal relationship' plus a thing
p	Preposition	Preposition	Expresses minimal relationship
cv	Completive	NP in PP	Expresses the thing

Quantity-quality group elements:

qqgp	Quantity-quality gp.	Adverbial or Adjectival	Realizes (quantities of) qualities
t	Temperer	Intensifier	Tempers the quality of the following adverb/adjective
a	Apex	Adjective or Adverb	The 'head' of the group
f	Finisher	–	Completes meaning of temperer

Participant roles: (Conflated with S and C1, C2)

Af	Affected	Patient
Ag	Agent	Actor
At	Attribute	
Ca	Carrier	
Cog	Cognizant	
Cre	Created	
Em	Emoter	
Loc	Location	
Perc	Perceiver	
Ph	Phenomenon	
Pos	Possessed	

Adjuncts bearing circumstantial roles:

Afreq	Frequency
Ahyp	Hypothetical
Ama	Manner
Apl	Place
Apol	Politeness
Areas	Reason
Atp	Time position
Ausu	Usuality