Sponsored by the Special Interest Group on Parsing of the Association for Computational Linguistics

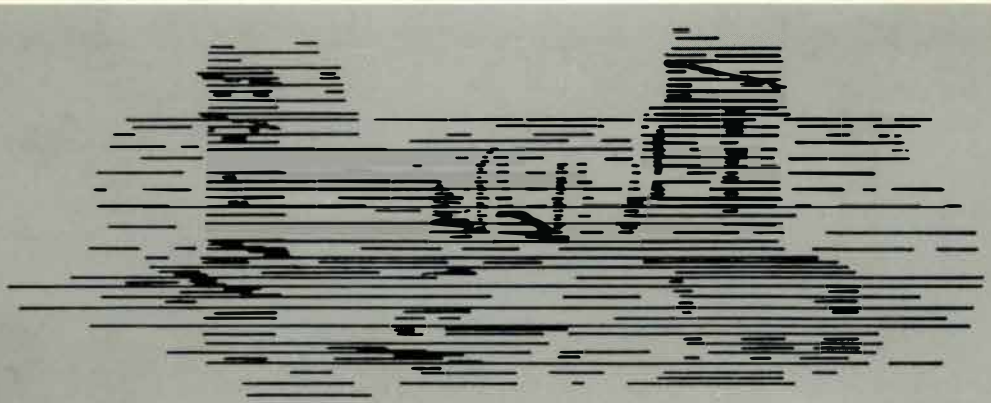
SIGPARSE

Proceedings

IWPT91

Second International Workshop on Parsing Technologies

13-15 February 1991

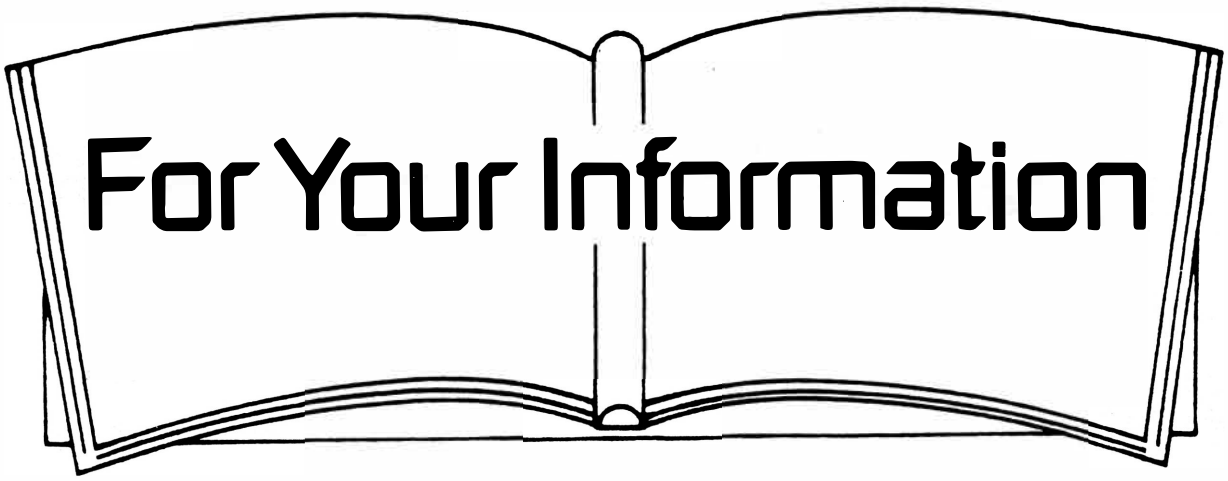


Sheraton Cancun
Resort & Towers
(Cancun, Mexico)

I W P T '91

Proceedings of the Second International Workshop on Parsing Technologies

February 13-25, 1991
Cancun, Mexico



For additional copies of these proceedings, write:

Joan Maddamma
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213-3890

Price: $50.00 USA - $55.00 for overseas

Proceeding designs and production by Joan Maddamma

Preface

WELCOME to the International Workshop on Parsing Technologies.

Parsing was the first topic in computational linguistics to separate itself from the relatively incoherent background of 'ad hoc' approaches to machine translation and information retrieval. While the emergence, in the early sixties, of general parsing algorithms that could take account of the ambiguity and nondeterminism that are apparently endemic to natural language was a cause for excitement, it would, I think have surprised those that experienced it to know how vigorous the field would remain after thirty years. This is due to a number of factors: Formal grammars and related processes have proved to be rich in mathematical properties, a fact that is attested to abundantly by the papers in this volume. It could certainly not have been foreseen that such a wide range of grammatical formalisms would emerge, partly as a response to transformational grammar. Furthermore, the notions of reversible, declarative, and psychologically plausible formalisms had yet to emerge. Statistical and connectionist approaches to grammar could only develop against a richer set of grammatical ideas.

All of these things have preserved the position of parsing as an exciting and continually changing field. And they have made the task of those responsible for the program of this conference especially difficult. I am therefore especially grateful to the program committee, Robert Berwick, Harry Bunt, Eva Hajicova, Aravind Joshi, Ronald Kaplan, Robert Kasper, Makoto Nagao, Masaru Tomita, and Yorick Wilks for their efforts. Masaru Tomita is, of course, especially to be thanked for realizing the need for a series of workshops of this kind, for arranging the first of them, and for his invaluable contributions to the planning of this one. In fact the only person to whom we owe a greater debt is Joan Maddamma, secretary to Prof. Tomita with whom I've developed a close working relationship: I made all the errors in the planning of the meeting, and she fixed them all up.

I look forward to a very exciting three days. I hope you will find the papers informative, the company stimulating, and the weather warm enough.

Martin Kay
IWPT '91 Program Chair

Workshop Committee

Program Committee:

Martin Kay, *Xerox PARC*
Chairman

Robert Berwick, *Massachusetts Institute of Technology*

Harry Bunt, *Tilburg University*

Eva Hajicova, *Charles University*

Aravind Joshi, *University of Pennsylvania*

Ronald Kaplan, *Xerox PARC*

Robert Kasper, *University of Ohio*

Masaru Tomita, *Carnegie Mellon University*

Makoto Nagao, *Kyoto University*

Yorick Wilks, *New Mexico State University*

Workshop Organizers

Masaru Tomita, Carnegie Mellon University
General Chairman

Martin Kay, Xerox Corporation
Program Chairman

Joan Maddamma, Carnegie Mellon University
Coordinator/Secretary

AUTHOR INDEX

TABLE OF CONTENTS

Session C.

February 14, 1991

Thursday - [9:00-2:00]

Session A.

Session B.

Session C.

Friday - February 15, 1991 - 9:00-2:00

Friday [9:00-2:00]

Session A.

Session B.

Session C.

February 13, 1991

Session A

PARSING WITHOUT PARSER

HASIDA, Kôiti TSUDA, Hiroshi*
Institute for New Generation Computer Technology (ICOT)
Mita Kokusai Bldg. 21F, 1-4-28 Mita, Minato-ku, Tokyo 108, JAPAN
Tel: +81-3-3456-3069
E-mail: hasida@icot.or.jp tsuda@icot.or.jp

ABSTRACT

In the domain of artificial intelligence, the pattern of information flow varies drastically from one context to another. To capture this diversity of information flow, a natural-language processing (NLP) system should consist of modules of constraints and one general constraint solver to process all of them; there should be no specialized procedure module such as a parser and a generator.

This paper presents how to implement such a constraint-based approach to NLP. *Dependency Propagation* (*DP*) is a constraint solver which transforms the program (=constraint) represented in terms of logic programs. *Constraint Unification* (*CU*) is a unification method incorporating DP. *cu-Prolog* is an extended Prolog which employs CU instead of the standard unification.

cu-Prolog can treat some lexical and grammatical knowledge as constraints on the structure of grammatical categories, enabling a very straightforward implementation of a parser using constraint-based grammars. By extending DP, one can deal efficiently with phrase structures in terms of constraints. Computation on category structures and phrase structures are naturally integrated in an extended DP. The computation strategies to do all this are totally attributed to a very abstract, task-independent principle: prefer computation using denser information. Efficient parsing is hence possible without any parser.

1 Introduction

The information-processing capacity of a cognitive agent is severely limited, whereas the world in which it finds itself contains a vast amount of information which might be relevant to its survival. A cognitive agent is thus destined to face *partiality of information*. That is, information processing by a cognitive agent is limited to a very small part of the potentially relevant information. In the domain of artificial intelligence in general and natural language processing in particular, therefore, the pattern of information flow varies very drastically from one context to another. This is necessary in order for a cognitive agent to have chance of access to the entire domain of potentially relevant information across various different contexts.

Due to this diversity of information flow, it is practically impossible to stipulate which pieces of information to process in which order. Consider the case of comprehension of natural language sentences, for instance. Parts of phonological information might be missing due to noise, and it may well be impossible to predict which part would be missing. Similarly, parts of syntactic information could be insufficient, giving rise to syntactic ambiguity. Semantic information also would be partially abundant or missing due to familiarity or ignorance to the topic, and so on. It is therefore utterly implausible to suppose that all phonological information is processed prior to syntactic information, or that syntactic information is processed before semantic information.

Accordingly, it is not at all a promising approach to AI or NLP to stipulate information flow totally, as procedural programs do. In particular, a hierarchical architecture consisting of modules of procedures fails to capture very complex, multi-directional, information flow in the domains such as NLP, because procedures stipulate what is input and what is output, severely restricting the global information flow across the entire system. This is what happens in the prevalent architecture of NLP systems consisting of a sequence of procedure modules such as, say, syntactic analyzer, semantic analyzer, pragmatic analyzer, generation planner, and surface generator.

The design of AI systems should abstract away information flow in accordance with its diversity.[1] This is where constraint paradigm [14] comes in. Since constraint, or declarative program, does not stipulate pro-

*The order is not significant.

[1]Of course, there are some aspects of cognitive process where information flow is rather restricted. Typical examples are found in low-level aspects of perception and motor control. Information flow may be stipulated to some adequate extent in the design of those subsystems. Nevertheless, diversity of information flow must be captured across different dimensions even in these cases, as is indicated by R. Brooks [1]; In his robot, although information flow in each module may be regarded as uni-directional and there is only a little interaction between different modules, input information is not restricted to flow all the way through every module before output information is tailored.

cessing order, it does not restrict information flow so severely as procedures do, and thus can capture the diversity of information flow.[2]

In the constraint paradigm, a NLP system involves modules of linguistic (syntactic, semantic, pragmatic, and so on.) and extralinguistic constraints. Whether there are different constraint solvers for different modules of constraints is not a light question, but we strongly suspect the answer is no. If yes, the communication between different modules would be too cumbersome to allow the massive interaction required in NLP. For instance, it would not be a very good idea to have a constraint solver specialized for processing syntactic information. Thus we employ a radical constraint-based viewpoint: just one very general constraint solver deals with all the different constraints, giving rise to diverse communication across them.[3] The task of NLP is hence divided into modules of constraints rather than modules of procedures as has been traditionally done. As a matter of course, a NLP system should include no parser.

In the rest of the paper, we will concentrate on parsing. Efficient parsing will be shown to emerge from our constraint solver, which is a general constraint transformation method employing several heuristics derived from the following very abstract, task-independent principle:

(1) Prefer computation using denser information.

That is, efficient parsing is attributed to this principle. This is regarded as an impressive demonstration of the feasibility of our constraint-based approach, because parsing is almost the only subproblem of NLP where there are endorsed efficient algorithms, mainly for dealing with phrase structures.

Our constraint solver, called *Dependency Propagation* [8, 7], deals with constraints in a combinatorial domain, unlike the constraint solvers embedded in most constraint logic programming (CLP) languages [2, 3, 9]. Section 2 describes how to parse ambiguous sentences with a CLP language, called *cu-Prolog*, which embeds an early version of DP. Although the ambiguity treated in Section 2 concerns only the structures of grammatical categories, Section 3 applies DP itself to parsing phrase-structure, typically formulated in terms of context-free grammars. It will be shown that efficient parsing procedures such as Earley's algorithm simply emerge from general processing strategies employed in a revised version of DP. Section 4 demonstrates that both types of information, about category structures and phrase structures, are processed efficiently in a naturally integrated manner by very general heuristics. Finally, Section 5 concludes the paper.

[2]Constraint is not the only approach to diversity of information flow. For instance, blackboard architecture is also regarded as aiming at the same thing. Coroutine implemented in languages such as CONNIVER [13] is another example. The reason why we employ constraint paradigm is twofold. First, it comes with intuitive declarative semantics. Second, it implements the diversity of information flow at finer-grained levels than might be captured in the other approaches.

[3]Thus we consider that General Problem Solver was basically on the right track. Its alleged failure was simply due to the immaturity of programming technologies.

2 Processing Category Structure

In [18], we introduced a symbolic CLP language cu-Prolog and showed how it applies to parsing based on JPSG (Japanese Phrase Structure Grammar) [5]. By treating grammatical principles and ambiguity concerning polysemy or homonymy straightforwardly in terms of constraints, syntactic, semantic and other types of ambiguity are processed in an integrated manner by *Constraint Unification* (*CU*). CU is the unifier employed in cu-Prolog, and is roughly regarded as the standard unification plus DP. cu-Prolog deals with various constraints on the structures of grammatical categories, without any special programming besides the encoding of the relevant constraints.

2.1 Dependency Propagation

For the sake of expository simplification, in this paper we restrict ourselves to Horn clauses, although DP is not actually so limited.

Dependency triggers constraint transformation in DP. Two occurrences of the same variable in a clause constitutes a *dependency* when both occurrences do not occupy any *vacuous argument place*. An argument place of an atomic formula is said to be *vacuous* when a variable filling that argument place is never instantiated by evaluating that atomic formula.

For instance, the first argument place of predicate `member` defined below is vacuous.

(2) a. `member(E,[E|_]).`
    b. `member(E,[_|S]) :- member(E,S).`

In the following clauses, (3) has no dependency, and (4) has a dependency because it is equivalent to (5).

(3) `:-member(a,X).`

(4) `:-member(X,[a,b,c]).`

(5) `:-member(X,Y),Y=[a,b,c].`

In DP, computation proceeds so as to eliminate dependency. Note that this is a more general control schema than Earley deduction [10], which executes the body of each clause in the fixed left-to-right order.

Basically, *fusion* replaces one or more literals with another, so as to eliminate dependency. Fusion is a sort of unfold/fold transformation for logic programs [15]. For example, `member(X,[a,b])` is replaced by `c0(X)`, where `c0` is a new predicate defined as follows.

(6) `c0(a).`
    `c0(b).`

That is, two atomic formulas, `member(X,Y)` and `Y=[a,b]`[4] have been fused to one atomic formula `c0(X)`.

The principle (1) provides us some heuristics for controlling fusion. For example, the elimination of dependency involving a variable binding, as in `p(a,X)`, should have higher priority than the elimination of dependency between two ordinary atomic formulas, as in `p(X), q(X)`. We will discuss heuristics further along this line later.

2.2 cu-Prolog

A program of cu-Prolog is a set of Constraint-Added Horn Clauses (CAHCs), A CAHC is a Horn Clause followed by constraints:

$$\overbrace{H}^{Head} :- \overbrace{B_1, B_2, \cdots, B_n}^{Body} ; \overbrace{C_1, C_2, \cdots, C_m}^{Constraint}.$$

The prolog part (the head plus the body) of a CAHC is processed procedurally just as in Prolog, whereas the constraint part is dynamically transformed with a sort of unfold/fold transformation during the execution of the former part. The following is the inference rule of cu-Prolog:

$$\frac{A, \mathbf{K};\mathbf{C}. \ , \ A' :\!- \mathbf{L};\mathbf{D}., \quad \theta = mgu(A, A'), \mathbf{C}' = dp(\mathbf{C}\theta, \mathbf{D}\theta)}{\mathbf{L}\theta, \mathbf{K}\theta; \mathbf{C}'}$$

A and A' are atomic formulas. $\mathbf{K}$, $\mathbf{L}$, $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{C}'$ are sequences of atomic formulas. $mgu(A, A')$ is the most general unifier between A and A'.

$dp(\mathbf{C})$ is a modular constraint that is equivalent to $\mathbf{C}$. If $\mathbf{C}$ is inconsistent, the application of the above inference rule fails because $dp(\mathbf{C})$ does not exist.

The following holds if $\mathbf{C}_i$ and $\mathbf{C}_j$ share no variable:

(7) $dp(\mathbf{C}) = dp(\mathbf{C}_1), \cdots, dp(\mathbf{C}_n)$.

For example,

(8) *dp(*`member(X,[a,b,c]), member(X,[b,c,d]), app(U,V)`*)*

returns a new constraint `c0(X),app(U,V)`, where the definition of `c0` is

```
(9) c0(b).
    c0(c).
```

but

(10) *dp(*`member(X,[a,b,c]),member(X,[k,l,m])`*)*

is not defined.

[4] `Y=[a,b]` may be further regarded as a bundle of five atomic formulas: `Y=[A|Z]`, `A=a`, `Z=[B|W]`, `B=b` and `W=[]`.

2.3 JPSG parser in cu-Prolog

In cu-Prolog, unification-based grammar such as HPSG or JPSG can be implemented naturally by treating the constraints formulated in those theories almost as they are. Figure 1 shows an example session of the JPSG parser when it processes an ambiguous sentence.[5] Below we discuss two examples of CAHC in the JPSG parser in cu-Prolog [18].

The first example concerns how to pack lexical ambiguity. The following is the lexical entry of a Japanese polysemic noun "hasi" that means bridge, chopsticks, or edge depending on contexts.

```
(11) lexicon(hasi,[...sem(TYPE,OBJ)]);
                  hasi_sem(TYPE,OBJ).
```

and predicate `hasi_sem` is defined as follows.

```
(12) hasi_sem(structure,bridge).
     hasi_sem(tool,chopsticks).
     hasi_sem(place,edge).
```

Constraint `hasi_sem(TYPE,OBJ)` represents various meanings of "hasi" and the ambiguity may be resolved during the parsing process when other constraints are imposed. Because such ambiguity is considered at one time, instead of divided into separate lexical entries, parsing process can be efficient.

In the second example, various feature principles of unification-based grammar are embedded in a phrase structure rule as constraints. The following clause shows the foot feature principle of JPSG: the foot feature value of the mother unifies with the union of those of her daughters.

```
(13) psr([ff(MS)],[ff(LDS)],[ff(RDS)]);
           union(LDS,RDS,MS).
```

`psr(Mother,Left_Daughter,Head)`
is a phrase structure rule followed by the constraint `union(LDS,RDS,MS)` which represents the foot feature principle. `MS`,`LDS`, and `RDS` are foot features of mother, left daughter, and right daughter respectively. The constraint is flexibly processed with the constraint transformation mechanism with a heuristic.

In traditional Prolog, these principles are supposed to be implemented in the following procedural way:

```
(14) psr([ff(MS)],[ff(LDS)],[ff(RDS)]) :-
           union(LDS,RDS,MS).
```

By applying this rule, `union(LDS,RDS,MS)` is executed immediately and parsing process may be inefficient when variables are not well instantiated. Note that it is practically impossible to stipulate the order to process linguistic constraints in advance.

[5] cu-Prolog is implemented in C language on UNIX 4.2/3BSD. This example is on SYMMETRY machine [19].

```
_:-p([ken,ga,ai,suru]).

v[Form_675, AJN{Adj_677}, SC{SubCat_679}]:SEM_681---[suff_p]
 |
 |--v[vs2, SC{p[wo]}]:[love,ken,Obj0_415]---[subcat_p]
 |   |
 |   |--p[ga]:ken---[adjacent_p]
 |   |   |
 |   |   |--n[n]:ken---[ken]
 |   |   |
 |   |   |__p[ga, AJA{n[n]}]:ken---[ga]
 |   |
 |   |__v[vs2, SC{p[ga], p[wo]}]:[love,ken,Obj0_415]---[ai]
 |
 |__v[Form_675, AJA{v[vs2,SC{p[wo]}]}, AJN{Adj_677}, SC{SubCat_679}]:SEM_681---[suru]

cat     cat(v, Form_675, [], Adj_677, SubCat_679, SEM_681)
cond     c7(Form_675, SubCat_679, Obj0_415, Adj_677, SEM_681)
True.
CPU time = 0.050 sec

_:-c7(F,SC,_,A,SEM).
  F = syusi  SC = [cat(p, wo, [], [], [], Obj00_30)]  A = []  SEM = [love,ken,Obj00_30];
  F = rentai  SC = []  A = [cat(n, n, [], [], [], inst(Obj00_38, Type3_36))]
  SEM = inst(Obj00_38, [and,Type3_36,[love,ken,Obj00_38]])
no.
CPU time = 0.017 sec
```

This is an example run of JPSG parser in cu-Prolog. The first line is a user's input. "Ken ga ai suru" has two readings: "Ken loves (someone)" and "(someone) whom Ken loves." The parser draws a parse tree and returns information (constraint) on the structure of the top node. In this example, the ambiguity of the sentence is captured as the two solutions of the piece of constraint `c7(F,SC,_,A,SEM)`. The first solution corresponds to "Ken loves (someone)." and the second solution "(someone) whom Ken loves."

Figure 1: Parsing an ambiguous sentence.

3 Processing Phrase Structure

The JPSG parser discussed in Section 2, however, cannot handle ambiguity on phrase structures because the parsing algorithm is written only in the Prolog part of CAHC. This section shows that chart parsing is naturally derived from a very general control strategy of an extended version of DP.

3.1 Context-Free Parsing by Fusion

Let us consider the following extremely simple context-free grammar.

(13) $P \rightarrow a$
$P \rightarrow PP$

The parsing of string $aa \cdots a$ under this grammar may be formulated in terms of the following constraint.[6]

```
(14) :- p(A^0,B), A^0=[a|A^1], ···, A^{n-1}=[a].
     p([a|X],X).
     p(X,Z) :- p(X,Y), p(Y,Z).
```

Note that the double occurrence of `Y` in the last clause does not count as a dependency, because the second argument place of `p` is vacuous. Thus the only dependency to eliminate now is that concerning A^0. Here, we replace $p(A^0,B)$ with $p_0(A^0)$, creating a new predicate p_0.

```
(15) :- p0(B), A^0=[a|A^1], ···, A^{n-1}=[a|A^n].
     p0(A^1).
     p0(Z) :- p(A^0,Y), p(Y,Z).
```

$p(A^0,Y)$ in the last clause is folded and we get

```
(16) p0(Z) :- p0(Y),p(Y,Z).
```

This clause has a dependency concerning `Y`. Then, the parsing process continues.

This transformation process is exempt from the infinite loop due to left recursion, unlike DCG of the standard type, because fusion includes some sort of tabulation technique [16]. If we had $A^0 = [b|A^1]$ instead of $A^0 = [a|A^1]$, for instance, we would have the following instead of (16).

```
(17) :- p0(B), A^0=[b|A^1], ··· .
     p0(Z) :- p0(Y), p(Y,Z).
```

Predicate p_0 lacks a finite proof, and hence is unsatisfiable under the minimal interpretation. This is detected by checking each predicate once when it is first given or created. Infinite loop is avoided in just the same manner also in a more complex case where every input symbol is a well-formed word but they are lined up in a wrong way.

In the current formulation, the computational complexity for processing context-free languages is exponential as to the sentence length. With respect to the above example, suppose that predicate r_i is such that for any assignment to variable X_i, there is a set of assignments to variables X^0 through X^{i-1} under which $r_i(X_i)$ is equivalent to the following:

(18) $p(A^0,X_0) \wedge p(X_0,X_1) \wedge \cdots \wedge p(X_{i-1},X_i)$

p_0 may be regarded as r_0. As it turns out, if a definition clause of r_i is (19) with $j = i$, then r_{i+1} will be created by fusion of $r_i(Y)$ and $p(Y, Z)$, whichever literal might be unfolded, and a definition clause of r_{i+1} will be (19) with $j = i + 1$.

(19) $r_j(Z)$:- $r_j(Y)$, $p(Y,Z)$.

Note that fusion of $r_j(Y)$ or $p(Y, Z)$ with any other literal never takes place, because `Y` is constrained nowhere else and the second argument place of `p` is vacuous. Since (29) is (19) with $j = 0$, it follows from induction on i that r_i is created during the current parsing for $0 < i < n$. A similar reasoning will prove that exponentially many corresponding predicates are created when the basic version of DP as described so far is applied to the following context-free grammar, because there are plural predicate symbols.

(20) $P \rightarrow a$ $\quad Q \rightarrow a$
$P \rightarrow PP$ $\quad Q \rightarrow PP$
$P \rightarrow PQ$ $\quad Q \rightarrow PQ$

3.2 Penetration

To remedy the inefficiency mentioned above, we revise DP by employing a different method of operation for constraint transformation. The new transformation operation we introduce here is *penetration*, which conveys information across clause boundaries.

For instance, consider clause (21), where predicate `p` is defined by (22).

```
(21) :- p(X,Y), p(X,Z), X=f(Y), Y=g(Z).
(22) p(f(A),A) :- s(A).
     p(C,a).
```

The information of `X=f(Y)` makes `X` penetrate through `p(X,Y)`, creating new predicate `q`, as follows:

```
(23) :- q(X,Y), p(X,Z), X=f(Y), Y=g(Z).
     q(X,A) :- X=f(A), q(A).
     q(X,a).
```

As indicated here, the first argument of `q` must always unify with `X` in the first clause, whereas its second argument has no such restriction.

[6] A^i represents the constant list of length `(n-i)` whose elements are "a"s.

In the following discussion, a penetrated variable is written with a superscript like X^1, and called a *transclausal variable*, which roughly corresponds to the global variable of programming languages such as Pascal and C. A transclausal variable may be treated as if it were a constant. Accordingly, a penetrated argument place are omitted for the sake of expository simplification. For instance, (23) may be rephrased as follows:

```
(24) :- q(Y), p(X¹,Z), X¹=f(Y), Y=g(Z).
     q(A) :- X¹=f(A), q(A).
     q(a).
```

Just as fusion, penetration has two cases: *unfolding* and *folding*. An unfolding, such as this case, introduces a new predicate, whereas a folding does not. Binding `X¹=f(B)` in the second clause unifies with `X¹=f(Y)`, and the resulting binding, `X¹=f(Y¹)`, is shared by the first and the second clause:

```
(25) :- q(Y), p(X¹,Z), X¹=f(Y¹), Y¹=g(Z).
     q(¹Y) :- X¹=f(Y¹), q(Y¹).
     q(a).
```

X^1 may penetrate through `p(X¹,Z)` as well:

```
(26) :- q(Y), q(Z), X¹=f(Y¹), Y¹=g(Z).
     q(Y¹) :- X¹=f(Y¹), q(Y¹).
     q(a).
```

This is a folding case of penetration.

A typical pattern of penetration is shown in Figure 2. `p(•,•)`s in the left-hand side of the figure all have the same sign, and those in the right-hand side all have the opposite sign. That is, either `p(•,•)`s in the left are all body literals and those in the right are all head literals, or vice versa. α represents a penetrating variable. We say that this penetration is *downward* in the former case, and *upward* in the latter. The penetration to get (23) and (26) is downward.

For $1 \leq i \leq n$, Ψ_i' is a duplication of Ψ_i except that `p(•,•)` has been replaced by `q(•,•)`. When Φ_i and Ψ_j are the same clause for some i and j, the situation will be more complicated in the sense that the duplication increases not only the right-hand half of the figure but also the left-hand half. The example shown in the next subsection includes some such cases.

As shown in the lower part of the figure, second or later penetration of α through the first argument of `p` is a folding, reusing `q` without introducing a new predicate. Corresponding unfolding and folding must be in the same direction: upward or downward. Otherwise the original combinations of clauses are not preserved. Suppose for instance that α is to penetrate through `p(•,•)` in Ψ_1 at the bottom stage in Figure 2. If we applied folding here, simply replacing this `p(•,•)` with a `q(•,•)`, the resulting configuration would lose the combination of Φ_3 and Ψ_1.

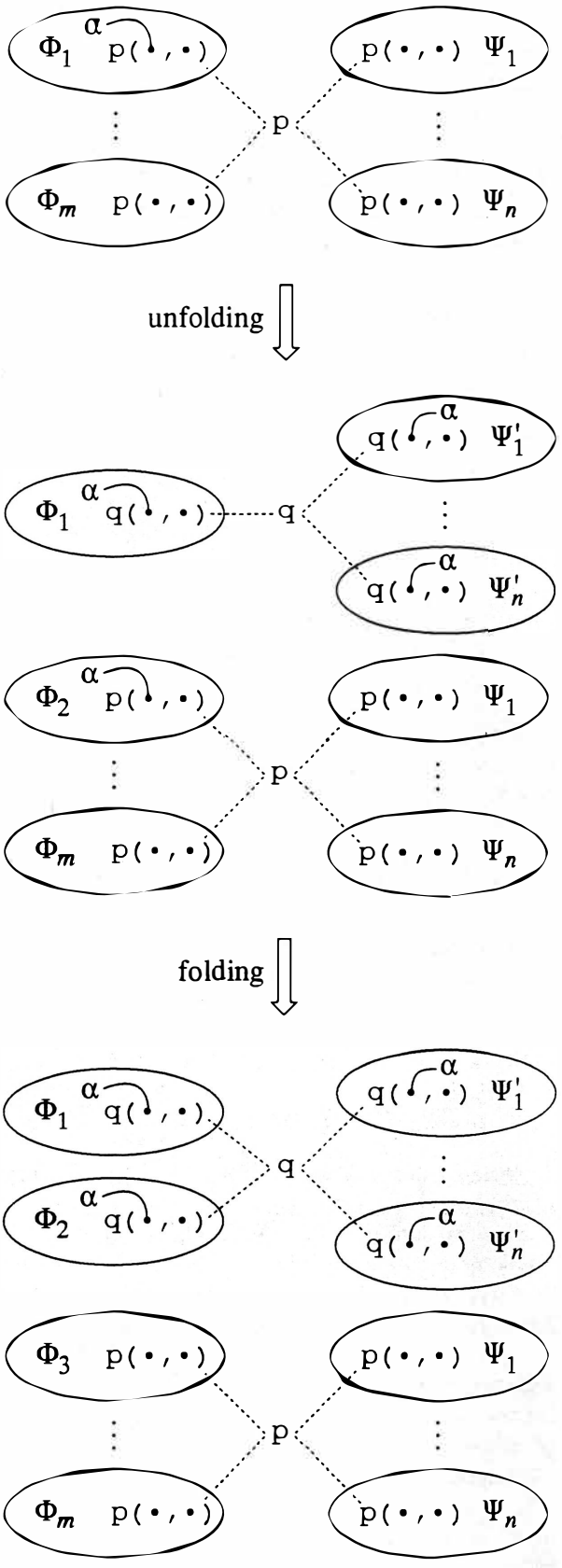


Figure 2: Penetration.

Like fusion, penetration is also triggered by dependency. In penetration, however, dependency may be transclausal. In (26), for instance, the dependency between `Y=g(Z)` and `q(Y)` could trigger penetration. This dependency is transclausal and involves a binding `Y=g(Z)`. In the case of upward penetration, the dependency in question involves a head literal.

To control computation, we must decide which dependency to trigger a penetration into which direction. The general principle (1) suggests the following heuristic in this respect.

(27) a. A dependency encompassing argument places with greater information quantity should more readily trigger a penetration.
b. The argument position with greater information quantity should be penetrated here.

(27b) guarantees that the resulting structure should have more homogeneous information distribution, increasing the entropy of the entire system.

For example, a binding in the top clause is considered to have much more information than bindings in the other clauses, in the sense that the atomic formulas in the top clause should primarily hold; if they do not, then we do not care whether the atomic formulas in the other clauses hold or not. The downward penetration occurring twice in the above example is motivated accordingly, because it is based on the information of `X=f(Y)` in the top clause.

3.3 Emergence of Chart Parsing

Now we demonstrate that Earley's algorithm naturally emerges from penetration controlled by the above heuristic. We consider the simple CFG example (13) again.

```
(13) :- p(A0,B), A0=[a|A1], ···, An-1=[a|An].
     p([a|X],X).
     p(X,Z) :- p(X,Y), p(Y,Z).
```

The following is obtained by downward penetration of A^0 through $p(A^0, B)$, which is unfolded.

```
(28) :- p0(B), A0=[a|A1], ···, An-1=[a|An].
     p0(A1).
     p0(Z) :- p(A0,Y), p(Y,Z).
```

The only relevant dependency here is the one concerning the first argument of $p(A^0, Y)$ in the bottom clause. This literal is hence folded and replaced with $p_0(Y)$, the entire clause being transformed as follows.

```
(29) p0(Z) :- p0(Y), p(Y,Z).
```

Now we have a non-vacuous dependency concerning `Y`, because p_0 says something substantial about the instantiation of its argument. The head $p_0(A^0)$ of the first definition clause of p_0 has transclausal variable A^0 as the argument. Since A^0 has been introduced in the top clause, upward penetration is applied here, so that the first definition clause of p_0 is replaced by $p_{0.1}$ .. and a new definition clause is introduced, as follows.

```
(30) p0.1 .
     p0(Z) :- p0.1, p(A1,Z).
     p0(Z) :- p0(Y), p(Y,Z).
```

The last clause of (28) has been replicated while $p_0(Y)$ therein has been replaced by $p_{0.1}$ plus $Y = A^1$, giving rise to the second clause in (30) above. Note that $p(Y)$ no longer imposes any restriction on the instantiation of `Y`. The dependency concerning `Y` in the third clause here is vacuous and left untouched for the time being.

A problem here, incidentally, is that another top clause as below is created.

```
(31) :- p0.1, B=A1, A0=[a|A1], ···,
                    An-1=[a|An].
```

To avoid two top clauses, we could introduce a new predicate `q` by which to mediate between the top clause and the locus of upward penetration:

```
(32) :- q, A0=[a|A1], ···, An-1=[a|An].
     q :- p0.1, B0=A1.
     q :- p0(B0).
```

Next, $p(A^1, Z)$ in the second clause of (30) is unfolded and a new predicate p_1 is created, A^1 penetrating downwards:

```
(33) p0(Z) :- p0.1, p1(Z).
     p1(A2).
     p1(Z) :- p1(Y), p(Y,Z).
```

Operation proceeds similarly, yielding the clauses below.

```
(34) p1,2 .
     p1(Z) :- p1,2, p2(Z).
     p0.2 :- p0,1, p1,2 .
     p0(Z) :- p0.2, p2(Z).
     p2(Z) :- p2(Y), p(Y,Z).
```

Shown below is what is finally obtained.

```
(35) :- q, A0=[a|A1], ···, An-1=[a|An].
     q :- p0(B0).
     q :- p0.i, B0=Ai. (0 < i ≤ n)
     pi(Z) :- pi.j, pj(Z). (0 ≤ i < j < n)
     pi(Z) :- pi(Y), p(Y,Z). (0 ≤ i < n)
     pi.i+1 . (0 ≤ i < n)
     pi.k :- pi.j, pj.k. (0 ≤ i < j < k < n)
```

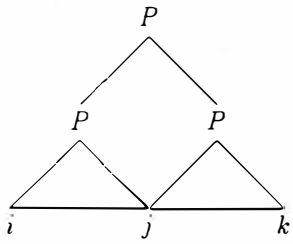


Figure 3: The meaning of $p_{i,k}$:- $p_{i,j}$, $p_{j,k}$.

3.4 Computational Complexity

Part of (35) amounts to a well-formed substring table, as in CYK algorithm, Earley's algorithm [4], chart parser, and so on. For instance, the existence of clause $p_{i,k}$:- $p_{i,j}$, $p_{j,k}$. means that, as illustrated in Fig. 3, the part of the given string from position i to position k has been parsed as having category P and is subdivided at position j into two parts, each having category P. Note that the computational complexity of the above process is $O(n^3)$ in terms of both space and time.

Moreover, the space complexity is reduced to $O(n^2)$ if we delete the literals irrelevant to instantiation of variables, which preserves the semantics of the constraints in the case of Horn programs. That is, the resulting structure would be:

(36) :- q, A^0=[a|A^1], $\cdots$, A^{n-1}=[a|A^n].
q :- $p_0(B^0)$.
q :- B^0=A^i. $(0 < i \leq n)$
p_i(Z) :- p_j(Z). $(0 \leq i < j < n)$
p_i(Z) :- p_i(Y), p(Y,Z). $(0 \leq i < n)$

Some sort of clauses listed here might be generated more than once in general cases where the grammar is less trivial than (13). For example, clause (37) may be derived from both (38) and (39).

(37) s_i(Z) :- vp_j(Z).

(38) s(X,Z) :- np(X,Y), vp(Y,Z).

(39) s(X,Z) :- np(X,Y), adv(Y,U), vp(Y,Z).

If (37) is generated twice, then of course we are able to collapse the two instances to one, so that the space complexity should be $O(n^2)$. Needless to say, this collapsing operation is totally domain-independent in its nature.

The process illustrated above corresponds best to Earley's algorithm. Our procedure may be generalized to employ more bottom-up control, so that the resulting process should be regarded as chart parsing in general, including left-corner parsing, and so on.

4 Integrated Processing

Section 2 treats linguistic constraints on category structures as constraint transformation, and Section 3 processed linguistic constraints on phrase structures. This section discusses how to handle various types of constraints mentioned in the previous two sections. Some heuristics will be needed to determine which constraint to process earlier than the others.

4.1 Heuristics

In the following discussion, we consider two types of linguistic constraints: constraints on category structure and those on phrase structure. For simplicity, the former constraints are represented only by predicate c, and the latter p. Accordingly, we introduce two types of dependency: *inter-dependency* and *intra-dependency*. Inter-dependency is a double occurrence of a variable in both types of constraints, such as X in p(X), c(X,Y). Intra-dependency arises with non-variable arguments or a variable that occurs only in one type of constraints such as c(a,X) or Y in c(a,Y),c(Y,b).

By applying the general heuristic (27) to this domain, we get the following heuristic:

- Eliminate intra-dependencies earlier than inter-dependencies.
- Eliminate intra-dependencies in category structure earlier than those in phrase structure.
- In eliminating inter-dependencies, the literal that has the fewer OR-alternatives should be unfolded (penetrated downward).

That is, constraints on category structures generally has more information quantity than those on phrase-structure, because the former are called by the latter. In the case of a dependency between two argument places of ordinary atomic formulas, moreover, penetration operation should take place at the one that has fewer alternatives of unfolding, because it is supposed to have more information quantity.

4.2 Example

The following is an ambiguous context free grammar that parses "I see a man with a telescope."

(40) $VP \rightarrow V\ NP$
$VP \rightarrow VP\ PP$
$NP \rightarrow NP\ NP$
$V \rightarrow$ see
$NP \rightarrow$ a man
$PP \rightarrow$ with a telescope

(41) is a parsing program in terms of this grammar.

```
(41) p(X,Z,C) :- p(X,Y,LC),p(Y,Z,RC),
                 c(LC,RC,C).
     c(v,np,vp).
     c(np,pp,np).
     c(vp,pp,vp).
     p([see|W],W,v).
     p([a,man|W],W,np).
     p([with,a,telescope|W],W,pp).
```

Predicate p represents phrase structure constraint and predicate c represents constraint on category structure.[7]

```
(42) :-p(A0,B,C), A0=[see|A1], A1=[a,man|A2],
                  A2=[with,a,telescope|A3],A3=[ ].
```

(42) is a question clause. This example shows that two meanings of "I see a man with a telescope" are derived from this program by the constraint transformation with the heuristic mentioned previously.

The dependency to be processed is in terms of A^0 in (42) because LC and RC in (40) do not have dependencies on account of vacuous argument places. Then, apply downward penetration in terms of A^0 to (42). $p_0(B,C)$ is equivalent to $p(A^0,B,C)$.

```
(43) :-p0(B,C).
(44) p0(A1,v).
(45) p0(B,C) :- p(A0,Y,LC),p(Y,B,RC),
                c(LC,RC,C).
```

The first body literal of (45) can be folded and we get

```
(46) p0(B,Cat) :- p0(Y,LC),p(Y,B,RC),
                  c(LC,RC,Cat).
```

Apply upward penetration to (44). Here $p_{0,1}$ is equivalent to $p_0(A^1,v)$.

```
(47) :-p0,1.
(48) p0,1.
(49) p0(B,Cat):-p0,1,p(A1,B,RC),c(v,RC,Cat).
```

Unfold the category constraint of (49).[8]

[7] From unification-based point of view, suppose each category has the form [pos/X] and c() represents the pos feature principle:

> The combination of the values of pos feature of mother, left daughter, and right daughter category is (vp,n,np), (np,np,pp), or (vp,vp,pp).

[8] Let c_0(Cat) be c(v,RC,Cat) and you apply downward penetration to (49), obtaining

$$p_0(B,Cat) :- p_{0,1},p(A^1,B,RC^1),c_0(Cat).$$

However, c_0 has only one definition clause:

$$c_0(vp):-RC^1=np.$$

So c_0 is reduced and you get (50).

```
(50) p0(B,vp):-p0,1,p(A1,B,np).
```

Now the remaining clauses are (43), (47), (48), (46) and (50). Apply downward penetration in terms of A^1 to (50). $p_1(B)$ is equivalent to $p(A^1,B,np)$.

```
(51) p0(B,vp):-p0,1,p1(B).
(52) p1(A2).
(53) p1(Z):-p(A1,Y,np),p(Y,Z,RC),c(np,RC,np).
```

Unfold the category structure constraint of (53).

```
(54) p1(Z):-p(A1,Y,np),p(Y,Z,pp).
```

The first body of (54) can be folded and we get

```
(55) p1(Z):-p1(Y),p(Y,Z,pp).
```

Upward penetration in (52). $p_{1,2}=p_1(A^2)$

```
(56) p0(A2,vp):-p0,1,p1,2.
(57) p1(Z):-p1,2,p1(A2,Z,pp).
(58) p1,2.
```

Upward penetration in (56). $p_{0,2}$ is equivalent to $p_0(A^2,vp)$.

```
(59) p0(B,Cat):-p0,2,p(A2,B,RC),c(vp,RC,Cat).
(60) p0,2:-p0,1,p1,2.
```

Unfold the category constraint of (59).

```
(61) p0(B,vp):-p0,2,p(A2,B,pp).
```

Here, the remaining clauses are (43), (47), (48), (58), (60), (46), (51), (55), (57) and (61). Apply downward penetration of A^2 in (61). $p_2(B)$ is equivalent to $p(A^2,B,pp)$.

```
(62) p0(B,vp):-p0,2,p2(B).
(63) p2(A3).
(64) p2(B):-p(A2,Y,LC),p(Y,B,RC),c(LC,RC,pp).
```

Unfolding of the category constraint of (64) fails. Fold (57).

```
(65) p1(Z):-p1,2,p2(Z).
```

Upward penetration in (63). $p_{2,3}$ is equivalent to $p_2(A^3)$.

```
(66) p0(A3,vp):-p0,2,p2,3.
(67) p2,3.
(68) p1(A3):-p1,2,p2,3.
```

Upward penetration in (66). $p_{0,3} \equiv p_0(A^3,vp)$.

```
(69) p0(B,Cat):-p0,3,p(A3,B,RC),c(vp,RC,Cat).
(70) p0,3.
```

Unfolding of the category constraint in (69) fails. Upward penetration in (70). $p_{1,3} \equiv p_1(A^3)$.

```
(71) p0(A3,vp):-p0,1,p1,3.
(72) p1(Z):-p1,3,p(A3,Z,pp).
(73) p1,3:-p1,2,p2,3.
```

(66) and (71) represent the two readings of "see a man with a telescope."

5 Concluding Remarks

In this paper, we have shown that various parsing techniques are subsumed in a general procedure of constraint transformation, whose control heuristic is attributed to an abstract, task-independent principle (1). Thus our conclusion is that no parser at all is needed in natural language processing, It is both desirable, as is discussed first in the paper, and possible, as we have so far demonstrated, for an NLP system to have no particular module for parsing sentences, just as a car has no particular part for driving towards the east or turning to the left.

Our approach will capture sentence generation as well, if we employ a more adequate control heuristic, which could also be derived from (1). In this connection, Shieber [12], among others, has also proposed a computational architecture by which to unify sentence parsing and generation, but his method is primarily specific to phrase-structure synthesis. A significant merit of our approach is that, as shown above, it is not in any way restricted to parsing or generation of context-free languages. Also, no additional mechanism is required to extend the underlying grammatical formalism so that grammatical categories may be complex feature bundles, as is the case with GPSG, LFG, HPSG, and so on.

At any rate, heuristics play the most important role in our approach. As this paper only gave an intuitive rationale on some heuristics in terms of information quantity, more formal account of them is yet to be worked out. A promising direction seems to be to define some sort of potential energy over constraints, which should capture information density, providing not only processing control but also preference of conclusion. Introducing hierarchies in the constraint is regarded as along the same line.

References

[1] Brooks, R. (1988) *Intelligence without Representation*, technical report, AI Laboratory, MIT.

[2] Colmerauer, A. (1987) *An Introduction to Prolog III*, unpublished manuscript.

[3] Dincbas, M., Simonis, H. and Van Hentenryck, P. (1988) 'Solving a Cutting-Stock Problem in Constraint Logic Programming,' *Proceedings of the 5th International Conference of Logic Programming*, pp. 42–58.

[4] Earley, J. (1970) 'An Efficient Context-Free Parsing Algorithm,' *Communications of ACM*, Vol. 13, pp. 94–102.

[5] Gunji, T. (1986) 'Japanese Phrase Structure Grammar', Reidel, Dordrecht,1986.

[6] Hasida, K. (1986) 'Conditioned Unification for Natural Language Processing,' *Proceedings of the 11th COLING*.

[7] Hasida, K. and Ishizaki, S. (1987) 'Dependency Propagation: A Unified Theory of Sentence Comprehension and Generation,' *Proceedings of the 10th IJCAI*, pp. 664–670.

[8] Hasida, K. (1990) 'Sentence Processing as Constraint Transformation,' *Proceedings of ECAI'90*.

[9] Jaffar, J. and Lassez, J. (1988) 'From Unification to Constraints,' *Logic Programming'87*, Lecture Notes in Computer Science, No. 315, pp. 1–18.

[10] Pereira, F.C.N. and Warren, D.H.D. (1983) 'Parsing as Deduction,' *Proceedings of ACL'83*, pp. 137–144.

[11] Pollard, C. and Sag, I.A. (1987) *Information-Based Syntax and Semantics. Volume 1*, CSLI Lecture Notes No. 13.

[12] Shieber, S.M. (1988) 'A Uniform Architecture for Parsing and Generation,' *Proceedings of the 12th COLING*, pp. 614–619.

[13] Sussman, G. and McDermott, D.V. (1972) *CONNIVER Reference Manual*, Memo 259, AI Laboratory, MIT.

[14] Sussman, G. and Steele, G., Jr. (1980) 'Constraints – A Language for Expressing Almost-Hierarchical Descriptions,' *Artificial Intelligence*, Vol. 14.

[15] Tamaki, H. and Sato, T. (1983) 'Unfold/Fold Transformation of Logic Programs,' *Proceedings of the Second International Conference on Logic Programming*, pp. 127–138.

[16] Tamaki, H. and Sato, T. (1984) 'OLD Resolution with Tabulation,' *Proceedings of the Third International Conference on Logic Programming*, pp. 84–98.

[17] Tsuda, H. and Hasida, K. (1990) 'Parsing as Constraint Transformation — an extension of cu-Prolog' *Proceedings of the Seoul International Conference on Natural Language Processing*, pp. 325–331.

[18] Tsuda, H., Hasida, K., and Sirai, H. (1989) 'JPSG Parser on Constraint Logic Programming,' *Proceedings of the European Chapter of ACL'89*, pp. 95–102.

[19] Tsuda, H., Hasida, K., Yasukawa,H. and Sirai, H. (1990) 'cu-Prolog V2 system', *ICOT TM-952*.

PARSING = PARSIMONIOUS COVERING?

Venu Dasigi
Department of Computer Science and Engineering
Wright State University Research Center
3171 Research Boulevard
Dayton, OH 45420
(513) 259-1395
(513) 873-3201
CS Net: vdasigi@cs.wright.edu

ABSTRACT

Many researchers believe that certain aspects of natural language processing, such as word sense disambiguation and plan recognition in stories, constitute abductive inferences. We have been working with a specific model of abduction, called *parsimonious covering*, applied in diagnostic problem solving, word sense disambiguation and logical form generation in some restricted settings. Diagnostic parsimonious covering has been extended into a dual-route model to account for syntactic and semantic aspects of natural language.

The two routes of covering are integrated by defining "open class" linguistic concepts, aiding each other. The diagnostic model has dealt with sets, while the extended version, where syntactic considerations dictate word order, deals with sequences of linguistic concepts. Here we briefly describe the original model and the extended version, and briefly characterize the notions of covering and different criteria of parsimony. Finally we examine the question of whether parsimonious covering can serve as a general framework for parsing.

1. Introduction

Natural languages are rife with ambiguity. There are lexical ambiguities; words in isolation may be seen to have multiple syntactic and semantic senses. There are syntactic ambiguities; the same sequence of words may be viewed as constituting different structures. And finally, there are semantic and pragmatic ambiguities, all of which may be resolved in context. Ambiguity and its context-sensitive disambiguation, it turns out, are two important characteristics of abductive inferences.

There have been various attempts at characterizing abductive inference and its explanatory nature [Appelt, 90; Charniak and McDermott, 85; Hobbs, et al., 88; Josephson, 90; Konolige, 90; Pople, 73; Reggia, 85; etc.]. While they differ somewhat in details, they all boil down to accounting for some observed features using potential explanations consistently in a "parsimonious" (often "minimal") way. Over the past decade, a formal model for abduction based on these ideas was developed at Maryland; this theory is called *parsimonious covering*. The theory originated in the context of simple diagnostic problems, but extended later for complex knowledge structures involving chaining of causal associations.

A diagnostic problem specified in terms of a set of observed manifestations is solved in parsimonious covering by satisfying the coverage goal and the goal of parsimony. Satisfying the coverage goal requires accounting for each of the observed manifestations through the known causal associations. Ambiguity arises here,

because the same manifestation may be caused by any one of several candidate disorders. Ensuring that a cover contains a "parsimonious" set of disorders satisfies the goal of parsimony. There could potentially be a large number of covers for the observed manifestations, but the "parsimonious" ones from among them are expected to lead to more plausible diagnoses. The plausible account for a manifestation may be one disorder in one context and another disorder in a different context. Such contextual effects are to be handled automatically by the specific criterion of parsimony that is chosen.

For medical diagnosis, reasonable criteria of parsimony are minimal cardinality, irredundancy and relevancy [Peng, 85]. Minimal cardinality says that the diagnosis should contain the smallest possible number of disorders that can cover the observed symptoms. A cover is considered irredundant (not redundant) if none of its proper subsets is also a cover, i.e., if the cover contains no disorder by removing which it can still cover the observed symptoms. Relevancy simply says that each disorder in the cover should be capable of causing at least one of the observed manifestations.

Consider an abstract example where disorder d_1 can cause any of the manifestations m_1 and m_2; d_2 can cause any of m_1, m_2 and m_3; d_3 can cause m_3; d_4 can cause m_3 and m_4; and finally, d_5 can cause m_4. If the manifestations $\{m_1, m_2, m_3\}$ were observed, the disorder set $\{d_2\}$ constitutes a minimal cardinality cover; the irredundant covers that are not minimal cardinality covers are $\{d_1, d_3\}$ and $\{d_1, d_4\}$; and an example of a redundant, but relevant cover would be $\{d_1, d_3, d_4\}$. While $\{d_2, d_5\}$ is a cover that has an irrelevant disorder (d_5) in it, $\{d_3, d_4\}$ is a non-cover, since together the disorders in this set cannot account for all observed manifestations.

Several natural language researchers have been actively involved in modeling abductive inferences that occur at higher levels in natural language, e.g., at the pragmatics level. Abductive unifications that are required in performing motivation analysis, for instance, might call for making the least number of assumptions that might potentially prove false [Charniak, 88]. Litman uses a similar notion of unification, called consistency unification [Litman, 85]. Hobbs and his associates propose a method that involves minimizing the cost of abductive inference where the cost might involve several different components [Hobbs, et al., 88]. Although [Charniak and McDermott, 85] indicate that word sense disambiguation might be viewed as abductive, nobody has pursued this line of research. It is very clear that there exists a strong analogy between diagnostic parsimonious covering and concepts in natural language processing. There are, however, important differences as well. These similarities and differences are summarized in Table 1. We have tried to extend parsimonious covering to address some of the idiosyncrasies of language (contrasted to diagnosis) and apply it to low level natural language processing.

2. Covering and Parsimony in Language

Linguistic concepts are viewed in parsimonious covering to be much like disorders and manifestations in diagnostic problems. However, in order to account for word order and structural constraints in language on the one hand and to account for the lexical and semantic content on the other, two aspects are attributed to each linguistic concept. These two aspects are loosely referred to as syntactic and semantic aspects, respectively. Concepts are covered parsimoniously in these two aspects, and the processes of covering are called syntactic and semantic covering.

TABLE 1: Similarities and Differences between Diagnostic Problem Solving and Natural Language Processing	
Parsimonious Covering Theory (Diagnosis)	Natural Language Processing
SIMILARITIES:	
symptoms	words
disorders	internal assertions
intermediate syndromes	word **senses** and structures
symptoms with multiple causes	ambiguous words
pathognomonic manifestations	unambiguous words
observed manifestations (to be explained)	input text (sequences of words) (to be interpreted)
causal relation (between symptoms and disorders)	lexical and semantic associations (between words and senses)
diagnostic explanation (i.e., a set of disorders)	semantic interpretation (i.e., a set of related assertions)
DIFFERENCES:	
order of entities ignored	word order important
sets of entities	sequences of concepts
only one type of knowledge (causal)	two types of knowledge (syntactic and semantic)

The notions of coverage and parsimony are briefly sketched here for syntactic covering through an abstract example here. Unlike in the case of diagnostic covering, the covers in syntactic covering are sequences rather than sets. Consider the following descriptions of categories c_1 through c_5 in terms of simpler categories (or words) w_0 through w_{10} below (sequences are indicated by being enclosed between "<>"):

c_1: <w_0 w_1 w_2 w_4 w_5 w_3 w_6>
c_1: <w_4 w_1 w_7 w_0>
c_2: <w_7 w_1 w_8>
c_3: <w_9 w_{10}>
c_4: <w_2 w_6 w_3>
c_5: <w_0 w_7>

The categories shown in **bold face** are mandatory categories, i.e., categories that **must** be present for the description to viably apply to a context. Semantic considerations govern whether a category is mandatory in a description. Depending on the domain, "the patient blind" might still make sense (indicating that the omitted copula is not mandatory), but "the patient" alone does not make complete sense (indicating that for this type of sentences, an adjectival complement is mandatory). See [Dasigi, 88] for discussion.

Suppose the input sequence is <w_1, w_2, w_3>. Some valid covers (covering sequences) are <c_1>, <c_1, c_3>, <c_3, c_1>, <c_2, c_4>, <c_2, c_3, c_4>, etc. Some non-covers are <c_2>, <c_4>, <c_2, c_3>, <c_4, c_2>,

etc., either because they cannot account for all the categories in the input sequence or because they cannot account for the correct order. Note that although <c_2, c_4> is a cover, <c_4, c_2> is not a cover. For instance, it makes sense to cover "paint the wall" with the sequence <Verb Noun-Phrase>, but not by <Noun-Phrase Verb>. Irredundant covers include <c_1> and <c_2, c_4>. Of these two irredundant covers, the former is also minimal (i.e., of minimal cardinality) and the latter is not. Insertion of c_5 into any valid cover causes it to be a non-viable cover since the category mandatory to c_5, namely, w_7 is not present in the input sequence to be covered. Thus, <c_1, c_5> is a non-viable cover.

Consider the cover <c_1, c_4>. Superficially, it appears to be a redundant cover since c_1 by itself is a cover. When the second rather than the first description of c_1 is taken into account, however, there is no redundancy in the cover, in a certain sense. For more concreteness, consider the following two classic sentences that differ in a single word:

"John painted **the wall with a crack.**"
"John painted **the wall with a brush.**"

Now, suppose there exist the usual descriptions for noun phrases (Noun-Phrase) and prepositional phrases (Prep-Phrase). Although in both sentences, the highlighted words can be syntactically covered by the irredundant cover <Noun-Phrase>, the sequence <Noun-Phrase Prep-Phrase> is a more appropriate cover in the second sentence, and we would like to consider that cover as irredundant, too. This characterization of irredundancy is obviously important, and is somewhat similar to the notion of "relevant diagnostic covers" defined in the previous section.

Semantic covering interacts closely with syntactic covering. Irredundant syntactic covering has a very nice property, namely, when complete sets of irredundant syntactic covers are considered, they are transitive across any number of layers when more than two layers of covering (e.g., as in typical parse trees) are involved [Peng and Reggia, 87; Dasigi, 88]. However, for a sequence of items, the number of irredundant covers at the next layer grows exponentially [Dasigi, 88]. Heuristics are needed for focusing search in such an ocean of covers, and semantic considerations serve this role. In the space of irredundant syntactic covers, search would be focused on "plausible" semantic covers.[1] Thus, the two routes of covering aid each other by syntactic covering providing a search space for semantic covering, and the latter focusing further syntactic covering at the next layer. Integration of the two routes of covering is facilitated by attributing *both* syntactic and semantic categories to distinguished linguistic concepts, called *open class* concepts.[2] In general, if the category that has just been postulated as a cover happens to be an open class category, it initiates semantic covering, thus integrating both the routes of covering.

3. Some Examples

A significant prototype was implemented to apply this algorithm in the context of an interface to an expert system. Instead of syntactic categories such as nouns, verbs, noun-phrases, etc., semantic categories were used in the syntactic covering process. Semantic covering was performed using domain-specific concepts defined in a knowledge base used by the expert system. In an OPS5-style expert

[1]Semantic covering also involves the notions of covering and parsimony, where parsimony considerations indicate the plausibility of semantic covers.

[2]This notion is very similar to that of open class words in languages. Non-open class concepts only have syntactic aspect, and correspond to "syntactic sugar" in language. See [Dasigi, 88] for more discussion.

system language, domain-specific concepts such as, **patient**, **vision**, **blind**, etc. were classified into semantic categories such as objects (**obj**), attributes (**attr**), values (**val**), etc. Two application domains were considered; the first domain is characterized by a sizable, prototype neurological knowledge base and the other deals with a toy chemical spills knowledge base. Some examples that were successfully handled by the prototype interfaces are:

"Visual acuity is blind."
"Visual acuity is blind on the left."
"Babinski on the left. Right unremarkable."
"The water is brown, radioactive and oily. Its pH is basic ..."

These examples demonstrate the use of lexical information, limited ability to handle ungrammatical sentences, interpretation of sentences in a discourse context rather than in isolation, etc. Note that the first few words of the first two inputs are the same. Their interpretations are, however, significantly distinct in the context of the knowledge base that was used, illustrating a form of non-monotonic inference in text interpretation. All but the last input is from the neurology domain and the last one is from the other.

A very simple example of parsimonious covering is given below to convey the flavor of the approach. Details are omitted due to space considerations, and we appeal to the reader's intuition in making sense out of this brief example. Suffice it to say that the category **assert** (and its variations) corresponds to sentences or clauses; **obj** and **attr** (and their variations) correspond to noun phrases; and **val** (and its variations) correspond to noun phrases or adjective complements. The category **asg-verb** stands for "assignment verb" (e.g., "is"). There are different ways an **assert** may be described in terms of the other categories mentioned so far. Often, **val** is a mandatory category in describing an **assert** (that is, it is unlikely that an **assert** makes semantic sense if a **val** is not present). Now, suppose a sentence begins with

"Vision is ..."

and is to be covered syntactically. One sequence of terminal categories that cover the first two words in this sentence is (**attr**, **asg-verb**) among others, since *vision* is an attribute and the word "is" is an instance of **asg-verb**. Since, this is an embedded sequence of what is expected of the above description of **assert**, the category **assert** is postulated to be a *non-viable syntactic cover* for the first two words. It is a *cover* because the two semantic categories occur in the description of **assert**, in the correct order. But the cover is *non-viable* nevertheless, because, not all mandatory categories in this particular description, namely, **val**, have occurred yet. When all expected mandatory categories occur, the cover will be considered viable. Further, viable or not, the cover is tentative because other possible covers exist and one of the other covers might prove to be globally more plausible. Now, suppose the sentence ends as follows:

... *impaired.*

Then, since *impaired* is a domain-specific value, the mandatory category is also encountered; so **assert** is confirmed as one of several viable syntactic covers for the given words. To keep things simple for the present purposes, it is assumed that **assert** turns out to be the most plausible syntactic cover.

The covering category in this example, namely **assert**, was designated as an open class category. In general, if the category that has just been postulated as a cover happens to be an open class category, it initiates semantic covering (with the standard notion of compositionality), thus integrating the use of both (that

is, syntactic and semantic) aspects of knowledge. Now, we continue the example from the viewpoint of semantic covering. Recall, however, that this process is interleaved with syntactic covering, and does not necessarily follow it. See Figure 1.

The word "vision" is covered, among other things indicated above, by a concept that has the semantic category **attr**. Category **attr** is of open class and so not surprisingly the concept that covers "vision" also has a domain-specific entity, say **a12**, that uniquely characterizes it. In effect, this one linguistic concept covering "vision," has two facets: the semantic category **attr** and the domain-specific entity **a12**. Similarly, the word "impaired" is covered by, among others, a concept of the semantic category **val** that has the unique domain-specific entity, say **v30**, associated with itself. The verb "is," however, is covered by a concept of the category **asg-verb** and since **asg-verb** is a not an open class category, it does not have a corresponding domain-specific entity.

As already explained in the course of syntactic covering, **assert** is computed to be a syntactic cover; it also turns out to be a *parsimonious* syntactic cover. For semantic covering, what needs to be covered is the set of entities grouped under this category, i.e., **a12** and **v30**, by identifying domain-specific associations that relate them. Definitions of parsimony and covering in the semantic route attempt to capture these intuitions, and the concept characterized by the semantic category **assert** and the domain-specific entity constituted by

(attr=a12, val=v30)

becomes the integrated parsimonious cover for the given sequence of words.

For the sake of completeness, we briefly describe the salient features of semantic covering. A detailed account and algorithms may be found in [Dasigi, 88]. The conceptual objects manipulated by semantic covering are domain-specific semantic senses. For semantic covering, the *order* of the concepts being covered is *no longer* important. Semantic covering involves discovering the relationships underlying the domain-specific entities evoked by input words, so that a parsimonious semantic cover can be synthesized for them; this cover corresponds to the logical form of the original sequence of words. There are two types of semantic covering. The first type of covering involves covering individual content words by domain-specific senses corresponding to objects, attributes, etc. This type of covering involves only lexical associations. Here, a domain-specific entity semantically covers a content word if any of the content words in the name or synonyms of the entity is morphologically related to the word itself or a domain-specific or domain-independent synonym of the word.

The other type of semantic covering is based on the relationships in a domain-specific semantic network. A simple domain-specific entity may be represented by a single node in the semantic network, e.g., an attribute. Also, a non-atomic subgraph of the semantic network can represent a more complex domain-specific entity, e.g., an assertion that relates an attribute and a possible value for it. Either kind of domain-specific entity - whether represented by a single node or by a subgraph in the domain-specific semantic network - is said to be covered by any of its supergraphs. Since any super-graph of a domain-specific concept can cover it, for any domain-specific concept there are potentially a huge number of covers, some of which are very redundant. There should be some means of controlling the number and sizes of potential covers. Criteria of parsimony and other constraints are used to achieve this control.

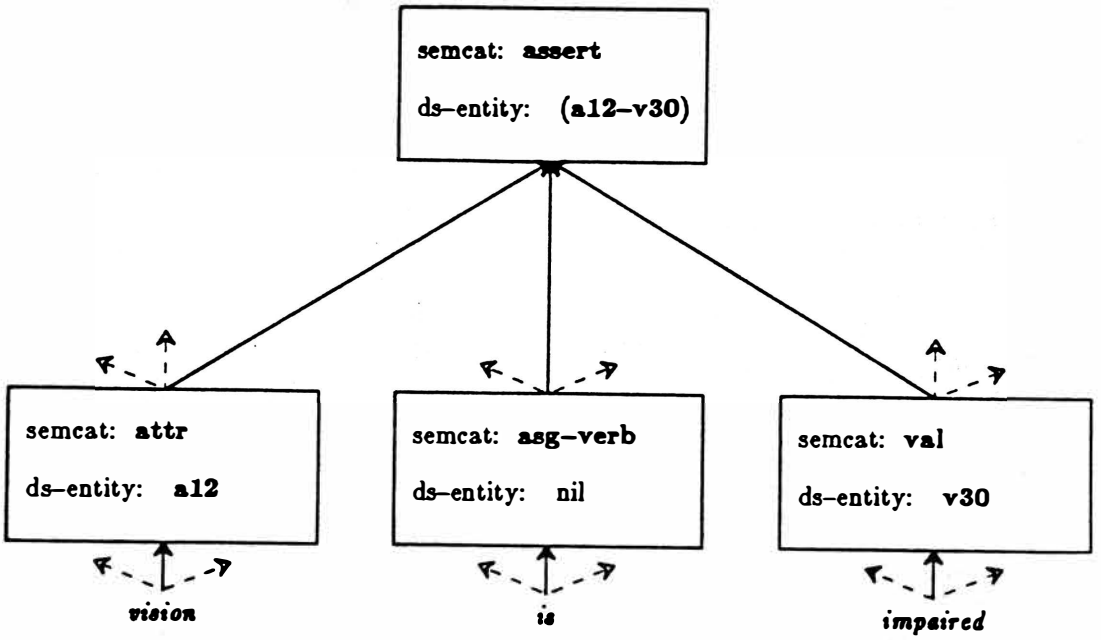


Figure 1: Interleaving of syntactic and semantic covering. The dashed arrows indicate other concepts that are evoked, e.g., other attributes named by "vision," other types of verbs that "is" evokes and many other concepts named by "impaired."

A criterion of parsimony called *cohesiveness* is chosen, inspired by the fact that in order to be understandable, text must be cohesively connected. A set of semantic categories are designated as assertionals (loosely corresponding to the notion of a sentence or an independent clause in English). A semantic cover corresponding to a non-assertional category is considered to be cohesive if it is the smallest (in terms of nodes) connected graph covering the concepts in question. A semantic cover corresponding to an assertional category is considered to be cohesive if either it is the smallest connected graph covering the concepts being covered or it is a not necessarily connected graph of several such domain-specific entities belonging to assertional categories. If there is more than one unconnected cover for the same concepts, the smallest connected cover of such unconnected components is the cohesive cover. It can be seen that cohesiveness refers to the "size" of the covers, and it is similar to "minimal cardinality," used in early versions of parsimonious covering theory for diagnostic problems. Indeed, if minimality were to be extended to structured entities, it would be similar to cohesiveness above. Cohesiveness refers to how well a cover fits into its surrounding context, a generalization of the notion of minimal cardinality, applied to structured entities.

Consider two consecutive concepts that have the same domain-specific entity (say an object) as one of the many candidate covers. Since both concepts can be covered by the same entity, the entity is a minimal cover for both of them together. This example of parsimonious covering is essentially the same as minimal covering in

the unextended parsimonious covering theory for diagnostic problem solving. However, suppose the two concepts involved *cannot* be covered by the same domain-specific entity. A minimal cover in the unextended parsimonious covering theory would consist of any pair of entities (pair - because there are two words to be covered) such that each entity in the pair covers one concept. But when structured entities with semantic associations among them are considered, the entities in the pair must also *unify*, taking domain-specific associations into account.[3] Unification of such structures corresponds to a search in the domain-specific semantic network, say, by marker passing [Charniak, 83].

One important remark about semantic covering is in order. Cohesiveness, as a notion of parsimony for semantic covering, is intended to capture how plausible a semantic cover is. But it is possible that a cohesive cover might turn out to be implausible when checked for well-formedness. Because of this possibility, there should be means to recompute the next most plausible (cohesive) cover. Thus, whenever a cohesive cover is found, all the irredundant covers must be saved so that the space of possibilities they constitute can be explored for cohesiveness if the cohesive cover that was found were to be rejected later. Consider the following abstract example. Let x_1, x_2, x_3, x_4 and x_5 be the senses of one ambiguous linguistic concept and y_1, y_2, y_3 and y_4 be the senses of another concept. If these two concepts were syntactically covered together by an open class semantic category, then semantic covering will be initiated. Now, what needs to be semantically covered is the conjunction of the following two disjunctions (representing 5*4 = 20 combinations):

$\{x_1\ x_2\ x_3\ x_4\ x_5\}$ and $\{y_1\ y_2\ y_3\ y_4\}$

Suppose a cohesive cover is found between x_2 and y_3. Then the irredundant cover will be constituted by the following three conjunctions of disjunctions (which represent the remaining 19 combinations):

$\{x_1\ x_3\ x_4\ x_5\}$ and $\{y_1\ y_2\ y_4\}$
$\{x_2\}$ and $\{y_1\ y_2\ y_4\}$
$\{x_1\ x_3\ x_4\ x_5\}$ and $\{y_3\}$

If the cohesive cover that was discovered gets rejected, the next most cohesive cover might be computed from these irredundant covers.

The dual-route parsimonious covering algorithm uses a discrete marker passing scheme to find cohesive semantic covers. One problem with irredundant syntactic covering is that typically there are too many such covers. (The advantage, however, is that all useful information is always available.) Since there are too many candidate syntactic covers, there exists a need to focus search for the best ones. Consequently, the dual-route algorithm uses semantic criteria to select a candidate to be covered at the next layer. Thus, the algorithm incorporates notions of parsimonious covering and best-first search to integrate syntactic and semantic processing towards the goal of synthesizing the final interpretation for an input text.

4. Discussion

The ability of parsimonious covering to handle ungrammatical sentences, as exemplified earlier, does not call for any special (or ad hoc) handling. It is a natural consequence of the very definition of covering itself. One could argue that a conventional production rule approach may

[3]This can be understood as follows: An assertion may be viewed as a predicate assert(?v,?a,?o), where ?v, ?a and ?o are variables such that ?v is a possible value of attribute ?a, which in turn is an attribute of object ?o. If one of the constituents is covered by a specific value v1 and the other is covered by a specific attribute a2, the covers effectively specify the assertions assert(v1,?aa,?oo) and assert(?vv,a2,?ooo), respectively. Now unification may be performed in the usual sense.

easily be augmented to achieve the same effect. For instance, it might be possible that a description such as:

assert: attr asg-verb val,

where **val** is mandatory, can be encoded into the following production rules:

assert --> attr asg-verb val
| attr val
| val
| ...,

the number of such rules can grow exponentially in the number of non-mandatory categories.

The previous paragraph should not be misconstrued as downplaying the significance of syntax in language. Indeed, the verb is plays a crucial role in disambiguating sentences such as,

"Flying planes is/are dangerous."

Our point is that omission of the copula in such sentences still does not make them incomprehensible. It does leave the sentence ambiguous, to be sure. At present, the semantic covering process does not worry about number agreement between the verb and subject, unless ambiguity arises. The underlying assumption here is that people try to make sense, and are not always grammatical.[1]

In summary, parsimonious covering provides a framework to view parsing natural language as an abductive process. A proof of concept is provided by implementing the basic ideas in an application independent interface shell. Admittedly, the semantic knowledge used is very restricted in nature, at the moment appropriate onlt to an object-oriented class of applications. The presumed logical form is also, correspondingly, of a limited generality. Many significant linguistic issues remain to be answered in this framework, however. Two features of this preliminary work (namely, use of a semantic grammar-like descriptions that are closely related to the class of expert systems for which interfaces could be generated, and reliance on the assumption that ambiguity resulting from ungrammaticality is resolvable in context) make it hard to predict the generality of the technique for unrestricted natural language. It is hoped that planned extensions, in the directions of using regular syntactic categories, and incorporation of further structure into verb definitions (consequently making the logical form much more general), might help answer these important questions.

[1]The majority of test inputs used by the prototype came from physicians' anonymous case descriptions, where insuring the grammaticality of sentences was, apparently, not the first priority.

Acknowledgements

The author acknowledges the support received from the State of Ohio Research Challenge grant that enabled him, in part, to prepare this paper. Past support from Jim Reggia of the University of Maryland is also gratefully acknowledged.

References

(1) Appelt, D., 90: A Theory of Abduction Based on Model preference, *AAAI Spring Symposium on Automated Abduction*, Stanford, March, 1990, pp. 67-71.

(2) Charniak, E., 83: Passing Markers: A Theory of Contextual Influence in Language Comprehension, *Cognitive Science*, 7(3), 1983, pp. 171-190.

(3) Charniak, E. and D. McDermott, 85: An Introduction to Artificial Intelligence. Addison Wesley, 1985, Chapters 8 and 10.

(4) Charniak, E., 88: Motivation Analysis, Abductive Unification and Nonmonotonic Equality, *Artificial Intelligence*, 34(3), 1988, pp. 275-295.

(5) Dasigi, V., 88: Word Sense Disambiguation in Descriptive Text Interpretation: A Dual-Route Parsimonious Covering Model. Ph.D. Dissertation. TR-2151, Department of Computer Science, University of Maryland, College Park, MD, 1988.

(6) Hobbs, J., M. Stickel, P. Martin and D. Edwards, 88: Interpretation as Abduction, *Proc. ACL-88,* 1988.

(7) Josephson, J., 90: On the "Logical Form" of Abduction, *AAAI Spring Symposium on Automated Abduction,* Stanford, March, 1990, pp. 140-144.

(8) Konolige, K., 90: A General Theory of Abduction, *AAAI Spring Symposium on Automated Abduction,* Stanford, March, 1990, pp. 62-66.

(9) Litman, D., 85: Plan Recognition and Discourse Analysis: An Integrated Approach for Understanding Dialogues. Ph.D. Dissertation, TR 170, Department of Computer Science, The University of Rochester, Rochester, NY 14627.

(10) Peng, Y., 85: A Formalization of Parsimonious Covering and Probabilistic Reasoning in Abductive Diagnostic Inference. Ph.D. Dissertation. TR-1615, Department of Computer Science, University of Maryland, College Park, MD 20742, January, 1986.

(11) Peng, Y. and J. Reggia, 87: Diagnostic Problem Solving with Causal Chaining, *International Journal of Intelligent Systems* 2, 1987, pp. 265-302.

(12) Pople, H., 73: On the Mechanization of Abductive Logic, *Advance Papers from the 3rd IJCAI,* Stanford, CA, 1973, pp. 147-152.

(13) Reggia, J., 85: Abductive Inference, *Proc. of IEEE Symposium on Expert Systems in Government,* Karna, K. N., (Ed). McLean, VA, 1985, pp. 484-489.

The Valid Prefix Property and Left to Right Parsing of Tree-Adjoining Grammar*

Yves Schabes
Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104-6389
schabes@linc.cis.upenn.edu

Abstract

The valid prefix property (VPP), the capability of a left to right parser to detect errors as soon as possible, often goes unnoticed in parsing CFGs. Earley's parser for CFGs (Earley, 1968; Earley, 1970) maintains the valid prefix property and obtains an $O(n^3)$-time worst case complexity, as good as parsers that do not maintain such as the CKY parser (Younger, 1967; Kasami, 1965). Contrary to CFGs, maintaining the valid prefix property for TAGs is costly.

In 1988, Schabes and Joshi proposed an Earley-type parser for TAGs. It maintains the valid prefix property at the expense of its worst case complexity ($O(n^9)$-time). To our knowledge, it is the only known polynomial time parser for TAGs that maintains the valid prefix property.

In this paper, we explain why the valid prefix property is expensive to maintain for TAGs and we introduce a predictive left to right parser for TAGs that does not maintain the valid prefix property but that achieves an $O(n^6)$-time worst case behavior, $O(n^4)$-time for unambiguous grammars and linear time for a large class of grammars.

*This research was partially funded by ARO grant DAAL03-89-C0031PRI and DARPA grant N00014-90-J-1863. The difficulty of maintaining the valid prefix property for TAGS was first noticed in joint work with Vijay-Shanker in the context of deterministic parsing of tree-adjoining grammars (Schabes and Vijay-Shanker, 1990). I am indebted to Vijay-Shanker for numerous discussions on this topic. I am also indebted to Aravind Joshi for his suggestions and for his support of this research. The discussions I had with Mitchell Marcus greatly improved the presentation of the algorithm introduced in this paper. I would also like to thank Bob Frank, Bernard Lang, Fernando Pereira, Philip Resnik and Stuart Shieber for providing valuable comments.

Organization of the paper

This paper discusses of two subjects: the difficulty of parsing tree-adjoining grammars (TAGs) while maintaining the valid prefix property (Sections 1 and 2) and the design of a predictive left to right parser for TAGs (Section 3). Although the two topics are related, they can be read independently of each other.

1 Definition of the Valid Prefix Property

The valid prefix property is a property of left of to right parsing algorithms which guarantees that errors in the input are detected "as soon as possible".

Parsers satisfying the *valid prefix property* guarantee that, as they read the input from left to right, the substrings read so far are valid prefixes of the language defined by the grammar: if the parser has read the tokens $a_1 \cdots a_k$ from the input $a_1 \cdots a_k a_{k+1} \cdots a_n$, then it is guaranteed that there is a string of tokens $b_1 \cdots b_m$ (b_i may not be part of the input) with which the string $a_1 \cdots a_k$ can be suffixed to form a string of the language; i.e. $a_1 \cdots a_k b_1 \cdots b_m$ is a valid string of the language.[1]

The valid prefix property is also sometimes referred as the error detecting property because it implies that errors can be detected as soon as possible. However, the lack of VPP does not imply that errors are undetected.

[1] The valid prefix property is independent from the *on-line* property. An on-line left to right parser is able to output for each new token read whether the string seen so far is a valid string of the language.

2 The Valid Prefix Property and Parsing of Tree-Adjoining Grammar

The valid prefix property, the capability of a left to right parser to detect errors as soon as possible, is often unobserved in parsing CFGs. Earley's parser for CFGs (Earley, 1968) maintains the valid prefix property and obtains a worst case complexity ($O(n^3)$-time), as good as parsers that do not maintain it, such as the CKY parser (Younger, 1967; Kasami, 1965). This follows from the path set complexity, as we will see.

Maintaining the VPP requires a parser to recognize the possible parse trees in a prefix order. The prefix traversal of the output tree consists of two components: a top-down component that expands a constituent to go to the next level down, and a bottom-up component that reduces a constituent to go to the next level up. When the VPP is maintained, these two components must be constrained together.

Context-free productions can be expanded independently of their context, in particular, independently of the productions that subsume them. The path set (language defined as the set of paths from root to frontier of all derived trees) of CFGs is therefore a regular set.[2] It follows that no additional complexity is required to correctly constrain the top-down and bottom-up behavior required by the prefix traversal of the parse tree: the expansion and the reduction of a constituent.

Contrary to CFGs, maintaining the valid prefix property for TAGs is costly.[3] Two observations corroborate this statement and an explanation can be found in the path set complexity of TAG.

Our first observation was that the worst case complexity of parsers for TAG that maintain the VPP is higher than the parsers that do not maintain VPP. Vijay-Shanker and Joshi (1985)[4] proposed a CKY-type parser for TAG that achieves $O(n^6)$-time worst case complexity.[5] As the original CKY parser for CFGs, this parser does not maintain the VPP. The Earley-type parser developed for TAGs (Schabes and Joshi, 1988) is bottom-up and uses top-down prediction. It maintains the VPP at a cost to its worst case complexity ($O(n^9)$-time in the worst case). However, the goal of our 1988 enterprise was to build a practical parser which behaves in practice better than its worst case complexity. Other parsers for TAGs have been proposed (Lang, 1988; Satta and Lavelli, 1990; Vijay-Shanker and Weir, 1990).[6] Although they achieve $O(n^6)$ worst case time complexity, none of these algorithms satisfies the VPP. To our knowledge, Schabes and Joshi's parser (1988) is the only known polynomial-time parser for TAG which satisfies the valid prefix property. It is still an open problem whether a better worst case complexity can be obtained for parsing TAGs while maintaining the valid prefix property.

The second observation is in the context of deterministic left to right parsing of TAGs(Schabes and Vijay-Shanker, 1990) where it was for the first time explicitly noticed that VPP is problematic to obtain. The authors were not able to define a bottom-up deterministic machine that satisfies the valid prefix property and which recognizes exactly tree-adjoining languages when used non-deterministically. Instead, they used a deterministic machine that does not satisfy the VPP, the bottom-up embedded push-down automaton, which recognizes exactly tree-adjoining languages when used non-deterministically.

The explanation for the difficulty of maintaining the VPP can be seen in in the complexity of the path set of TAGs. Tree-adjoining grammars generate some languages that are context-sensitive. The path set of a TAG is a context-free language (Weir, 1988) and is therefore more powerful than the path set of a CFG. Therefore in TAGs, the expansion of a branch may depend on the parent super-tree, i.e. what is above this branch. Going bottom-up, these dependencies can be captured by a stack mechanism since trees are embedded by adjunction. However, if one would like to maintain the valid prefix property, which requires traversing the output tree in a prefix fashion, the dependencies are more complex than a context-free language and the complexity of the parsing algorithm increases.

For example, consider the trees α, β and γ in Figure 1. When γ is adjoined into β at the B node, and the result is adjoined into α at the A node, the resulting tree yields the string $ux'zx"vy"ty'w$ (see Figure 1).

[2] This result follows from Thatcher's work (1971), which defines frontier to root finite state tree automata.

[3] We assume familiarity with tree-adjoining grammars. See, for instance, the introduction by Joshi (Joshi, 1987).

[4] The parser is also reported in Vijay-Shanker (1987).

[5] However, this algorithm is not a practical parser for TAGs because, as is well known for CFGs, the average behavior of CKY-type parsers is the same as the worst case behavior.

[6] In a recent paper, Karen Harbusch (1990) claimed to have defined an $O(n^4 log(n))$ worst time general TAG parser based on the CKY parser for CFGs. However, since the paper does not include a proof of correctness and complexity, the relationship between the parser and the set of

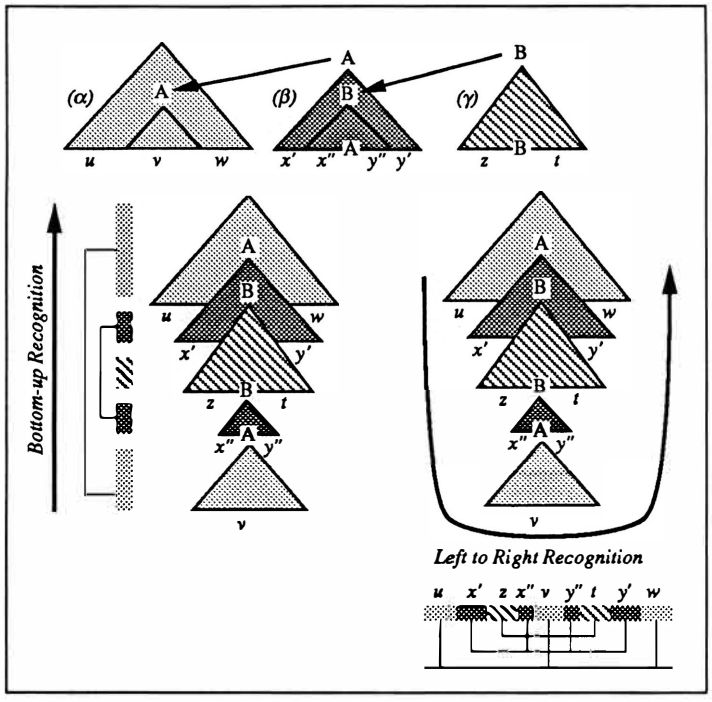


Figure 1: *Above*, a sequence of adjunctions; *below left*, bottom-up recognition of the derived tree; *right*, left to right recognition of the derived tree.

If this TAG derived tree is recognized purely bottom-up from leaf to root (and therefore without maintaining the VPP), a stack based mechanism suffices for keeping track of the trees to which to algorithm needs to come back. This is illustrated by the fact that the tree domains are embedded (see bottom left tree in Figure 1) when they are read from leaf to root in the derived tree.

However, if this derivation is recognized from left to right while maintaining the valid prefix property, the dependencies are more complex and can no longer be captured by a stack (see bottom right tree in Figure 1).

The context-free complexity of the path set of TAGs makes the valid prefix property costly to maintain. We suspect that the same difficulty arises for context-sensitive formalism which use operations such as adjoining or wrapping (Joshi et al., Forthcoming 1990).

languages it recognizes still needs to be determined.

3 A Predictive Left to Right Parser for TAGs

In this section, we define a new predictive left to right (Earley-style) parser for TAGs with adjoining constraints (Joshi, 1987). It is a bottom-up parser that uses some but not all the top-down information given by prediction. As a consequence, the parser does not satisfy the valid prefix property: it always detects errors but not as soon as possible. However, it achieves an $O(n^6)$-time worst case behavior, $O(n^4)$-time for unambiguous grammars and linear time for a large class of grammars (for example, the language $a^n b^n e c^n d^n$ is recognized in linear time). This parser as well as in the one introduced by Schabes and Joshi (1988) are practical parsers for TAGs since as is well known for CFGs, the average behavior of Earley-style parsers is superior to their worst case complexity. The algorithm has been modified to handle extensions of TAGs such as substitution, feature structures for TAGs and a version of multiple component TAG (these extensions are explained in Schabes [1990]).

3.1 Preliminary Concepts

Any *tree* α will be considered to be a function from tree addresses to symbols of the grammar (terminal and non-terminal symbols): if x is a valid address in α, then $\alpha(x)$ is the label of the node at address x in the tree α. *Addresses* of nodes in a tree are encoded by Gorn-positions (Gorn, 1965) as defined by the following inductive definition: 0 is the address of the root node, k ($k \in \mathcal{N}$) is the address of the k^{th} child of the root node, $x \cdot y$ (x is an address, $y \in \mathcal{N}$) is the address of the y^{th} child of the node at address x.

Given a tree α and an address *address* in α, we define $Adjunct(\alpha, address)$ to be the set of auxiliary trees that can be adjoined at the node at address *address* in α. For TAGs with no constraints on adjunction, $Adjunct(\alpha, address)$ is the set of elementary auxiliary trees whose root node is labeled by $\alpha(address)$.

We define a *dotted tree* as a tree associated with a dot *above* or *below* and either *to the left* or *to the right* of a given node. The four positions of the dot are annotated by la, lb, ra, rb (resp. left above, left below, right above, right below): ${}^{la}_{lb}A^{ra}_{rb}$. We write $< \alpha, dot, pos >$ for a dotted tree in which the dot is at address *dot* and at position *pos* in the tree α.

The *tree traversal* we define for parsing TAGs consists of moving a dot in an elementary tree in a manner consistent with the left to right scanning

of the yield while still being able to recognize adjunctions on interior nodes of the tree. The tree traversal starts when the dot is above and to the left of the root node and ends when the dot is above and to the right of the root node. At any time, there is only one dot in the dotted tree. An example of tree traversal is shown in Figure 2.

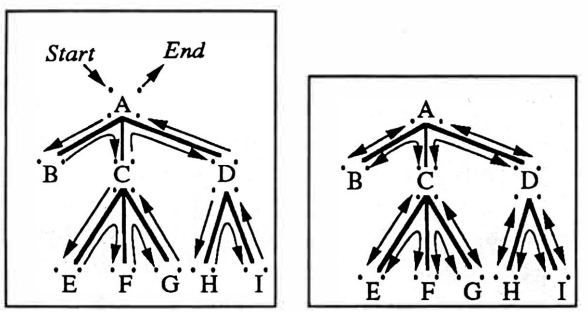


Figure 2: *Left*, left to right tree traversal; *right*, equivalent dot positions.

This traversal will enable us to scan the frontier of an elementary tree from left to right while trying to recognize possible adjunctions between the above and below positions of the dot.

We consider to equivalent two successive (according to the tree traversal) dot positions that do not cross a node in the tree (see Figure 2). For example the following equivalences hold for the tree α pictured in Figure 2: $< \alpha, 0, lb > \equiv < \alpha, 1, la >$, $< \alpha, 1, ra > \equiv < \alpha, 2, la >, < \alpha, 2, lb > \equiv < \alpha, 2 \cdot 1, la >, \cdots$.

3.2 Data Structures

We now define the data structures used by the parser. The input string is $a_1 \cdots a_n$ and the tree-adjoining grammar is $G = (\Sigma, NT, I, A)$: Σ is the finite set of terminal symbols, NT is the set of non-terminal symbols ($\Sigma \cap NT = \emptyset$), I is the set of initial trees and A is the set of auxiliary trees.

The algorithm uses one data structure: a state.[7]

A state s is defined as an 8-tuple, $s = [\alpha, dot, pos, i, j, k, l, sat?]$ where:

- α is an elementary tree, initial or auxiliary tree: $\alpha \in I \cup A$.
- dot is the address of the dot in the tree α.
- pos is the position of the dot: to the left and above, or to the left and below, or to the right and below, or to the right and above; $pos \in \{la, lb, rb, ra\}$.
- i, j, k, l are indices of positions in the input string ranging over $\{0, \cdots, n\} \cup \{-\}$, n being the length of the input string and $-$ indicating that the index is not bound.
- $sat?$ is a boolean; $sat? \in \{true, nil\}$.

The components α, dot, pos of a state define a dotted tree. Similarly to a dotted rule for context-free grammars defined by Earley (1968), a dotted tree splits a tree into two contexts: a left context that has been traversed and a right context that needs to be recognized.

The additional indices i, j, k, l record the portions of the input string.

The boolean $sat?$ indicates whether an adjunction has been recognized on the node at address dot in the tree α.

In the following, we will refer to one of the two equivalent dot positions for the dotted tree. For example, if the dot at address dot and at position pos in the tree α is equivalent to the dot at address dot' at position pos' in α, then $s = [\alpha, dot, pos, i, j, k, l, sat?]$ and $s' = [\alpha, dot', pos', i, j, k, l, sat?]$ refer to the same state. We will use to our convenience s or s' to refer to this unique state.

3.3 Analogy between Dotted Trees and Dotted Rules

There is a useful analogy between dotted TAG trees and dotted rules. It is by no mean a formal correspondence between TAG and a production system but it will give an intuitive understanding of the parser we define. It will also be used as a notation for referring to a dotted tree.

One can interpret a TAG elementary tree as a set of productions on pairs of trees and addresses (i.e. nodes). For example, the tree in Figure 2, let's call it α, can be written as:

$$(\alpha, 0) \rightarrow (\alpha, 1)\ (\alpha, 2)\ (\alpha, 3)$$
$$(\alpha, 2) \rightarrow (\alpha, 2 \cdot 1)\ (\alpha, 2 \cdot 2)\ (\alpha, 2 \cdot 3)$$
$$(\alpha, 3) \rightarrow (\alpha, 3 \cdot 1)\ (\alpha, 3 \cdot 2)$$

Of course, the label of the node at address i in α is associated with each pair (α, i).[8] One can then relate a dotted tree to a dotted rule. For example, consider the dotted tree $< \alpha, 2, ra >$ in which the dot is at address 2 and at position "right above" in the tree α (tree in Figure 2). Note that the dotted trees $< \alpha, 2, ra >$ and $< \alpha, 3, la >$ are equivalent.

[7] We could have chosen to group the states into state sets as in (Earley, 1968) but we chose not to, allowing us to define an agenda driven parser.

[8] TAGs could be defined in term of such productions. However adjunction must be defined within this production system. This is not our goal, since we want to draw an analogy and not to define a formal system.

The dotted tree $< \alpha, 2, ra >$ is analogous to the dotted rule:

$$(\alpha, 0) \rightarrow (\alpha, 1)\ (\alpha, 2) \bullet (\alpha, 3)$$

One can therefore put into correspondence a state defined on a dotted tree with a state defined on a dotted rule. A state $s = [\alpha, dot, pos, i, j, k, l, sat?]$ can also be written as the corresponding dotted rule associated with the indices i, j, k, l and the flag $sat?$:

$\eta_0 \rightarrow \eta_1 \cdots \eta_y \bullet \eta_{y+1} \cdots \eta_z\ [i, j, k, l, sat?]$
where $\eta_0 = (\alpha, u)$ *and* $\eta_p = (\alpha, u \cdot p),\ p \in [1, z]$

Here u is the address of the parent node of the dotted node when the dot is above to the left or to the right, and where $u = dot$ when the dot is below to the left or to the right.

3.4 Invariant of the Algorithm

The algorithm collects states into a set $\mathcal{C}$ called a *chart*. The algorithm maintains an invariant that is satisfied for all states in the chart $\mathcal{C}$. The correctness of the algorithm is a corollary of this invariant. The invariant is pictured in Figure 3 in terms of dotted trees. We informally describe it equivalently in terms of dotted rules. A state of the form:

$\eta_0 \rightarrow \eta_1 \cdots \eta_y \bullet \eta_{y+1} \cdots \eta_z\ [i, j, k, l, sat?]$
with $\eta_0 = (\alpha, u)$ *and* $\eta_p = (\alpha, u \cdot p)$

is in the chart if and only if the elementary tree α derives a tree such that:

(1) $\eta_1 \cdots \eta_y \stackrel{*}{\Rightarrow} a_i \cdots a_l$
(2) $(\alpha, f) \stackrel{*}{\Rightarrow} a_{j+1} \cdots a_k$

where f is the address of the foot node of α. (2) only applies when the foot node of α (if there is one) is subsumed by one of the nodes $\eta_1 \cdots \eta_y$

When the $pos = rb$, the dot is at the end the dotted rule and if $sat? = t$ the boundaries $a_i \cdots a_l$ include the string introduced by an adjunction on the dotted tree.[9]

$sat? = t$ indicates that an adjunction was recognized on the dotted node (node at address dot in α). No more adjunction must be attempted on this node.[10]

[9] The algorithm will set $sat?$ to t only when $pos = rb$.

[10] The derivations in TAG disallow more than one auxiliary tree adjoined on the same node.

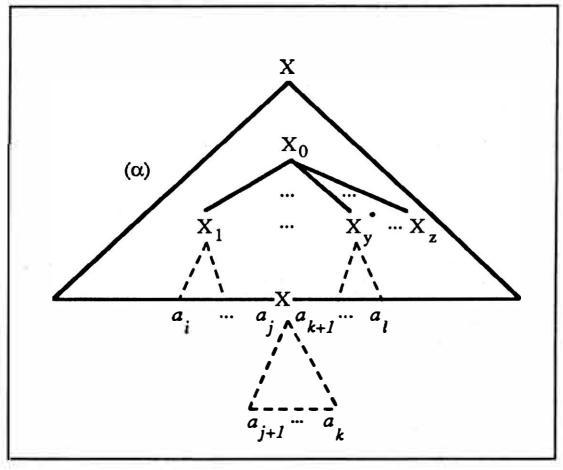


Figure 3: Invariant.

3.5 The Recognizer

The algorithm is a bottom-up parser that uses top-down information. It is a general recognizer for TAGs with adjunction constraints. Unlike the CKY-type algorithm for TAGs, it requires no condition on the grammar: the elementary trees (initial or auxiliary) need not to be binary and they may have the empty string as frontier. The algorithm given below is off-line: it needs to know the length n of the input string before starting any computation. However it can very easily be modified to an on-line algorithm by the use of an end-marker in the input string.

Initially, the chart $\mathcal{C}$ consists of all states of the form $[\alpha, 0, la, 0, -, -, 0, nil]$, with α an initial tree. These initial states correspond to dotted initial trees with the dot above and to the left of the root node (at address 0).

Depending on the existing states in the chart $\mathcal{C}$, new states are added to the chart by four processors until no more states can be added to the chart. The processors are: the *Predictor*, the *Completor*, the *Adjunctor* and the *Scanner*. If, in the final chart, there is a state corresponding to an initial tree completely recognized, i.e. with the dot to the right and above the root node which spans the input form position 0 to n (i.e. a state of the form $[\alpha, 0, ra, 0, -, -, n, nil]$), the input is recognized.

The recognizer for TAGs follows:

```
Let G = (Σ, NT, I, A) be a TAG.
Let a1 ··· an be the input string.

program recognizer
begin
  C = {[α, 0, la, 0, −, −, 0, nil]| α ∈ I }
```

Apply one of the four processes on each state in the $\mathcal{C}$ until no more states can be added to the $\mathcal{C}$:

1. Scanner
2. Predictor
3. Completor
4. Adjunctor

If there is a state of the form $[\alpha, 0, ra, 0, -, -, n, nil]$ in $\mathcal{C}$ with $\alpha \in I$ then return acceptance otherwise return rejection .

end.

The initialization step

$\mathcal{C}$={ $[\alpha, 0, la, 0, -, -, 0, nil] | \alpha \in I$ }

puts all initial trees with the dot to the left and above the root node. The four processes are explained one by one in the four next sections.

3.5.1 Scanner

The *Scanner* is a bottom-up processor that scans the input string. It applies when the dot is to the left and above a terminal symbol. It consists of two cases: one when the terminal is not the empty string, and the other when it is.

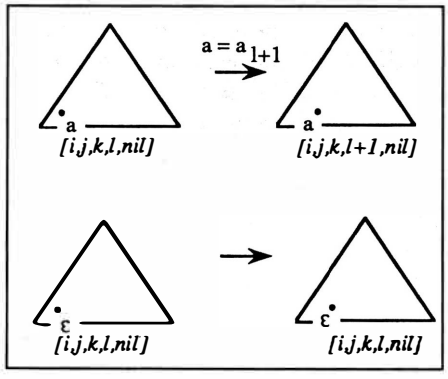


Figure 4: Scanner.

Scanner (see Figure 4):

It applies to a state of the form $s = [\alpha, dot, la, i, j, k, l, nil]$ such that $\alpha(dot) \in \Sigma \cup \{\epsilon\}$.

- Case 1: $\alpha(dot) = a_{l+1}$
  The state $s = [\alpha, dot, ra, i, j, k, l+1, nil]$ is added to $\mathcal{C}$.
- Case 2: $\alpha(dot) = \epsilon$ (empty string)
  The state $s = [\alpha, dot, ra, i, j, k, l, nil]$ is added to $\mathcal{C}$.

3.5.2 Predictor

The *Predictor* constitutes the top-down processor of the parser. It predicts new states accordingly to the left context that has been been read.

The *Predictor* creates new states from a given state. It consists of three steps which are not applicable all simultaneously. Step 1 applies when the dot is to the left and above a non-terminal symbol. All auxiliary trees adjoinable at the dotted node are predicted. Step 2 also applies when the dot is to the left and above a non-terminal symbol. If there is no obligatory adjoining constraint on the dotted node, the algorithm tries to recognize the tree without any adjunction by moving the dot below the dotted node. Step 3 applies when the dot is to the left and below the foot node of an auxiliary tree. The algorithm then considers all nodes on which the auxiliary tree could have been adjoined and tries to recognize the subtree below each one.

It is in Step 3 of the predictor that the VPP is violated since not all nodes that are predicted are compatible with the left context seen so far. The ones that are not compatible will be pruned in a later point in the algorithm (by the Completor). Ruling them out during this step requires more complex data-structures and increases the complexity of the algorithm (Schabes and Joshi, 1988).

Predictor (see Figure 5):

- Step 1 applies to a state of the form $s = [\alpha, dot, la, i, j, k, l, nil]$ such that $\alpha(dot) \in NT$.

  If the conditions are satisfied, the states $\{[\beta, 0, la, l, -, -, l, nil] | \beta \in Adjunct(\alpha, dot)\}$ are added to $\mathcal{C}$.

- Step 2 applies to a state of the form $s = [\alpha, dot, la, i, j, k, l, nil]$ such that $\alpha(dot) \in NT$ and the node at address dot in α has no obligatory adjoining constraint.

  If the conditions are satisfied, the state $[\alpha, dot, lb, i, j, k, l, nil]$ is added to $\mathcal{C}$.

- Step 3 applies to a state of the form $s = [\alpha, dot, lb, i, j, k, l, nil]$ such that such that $\alpha \in A$, and such that dot is the address of the foot node of α.

  If the conditions are satisfied, for all elementary trees $\delta \in I \cup A$ and for all addresses dot' in δ such that $\alpha \in Adjunct(\delta, dot')$, the state $[\delta, 0, lb, l, -, -, l, nil]$ is added to $\mathcal{C}$.

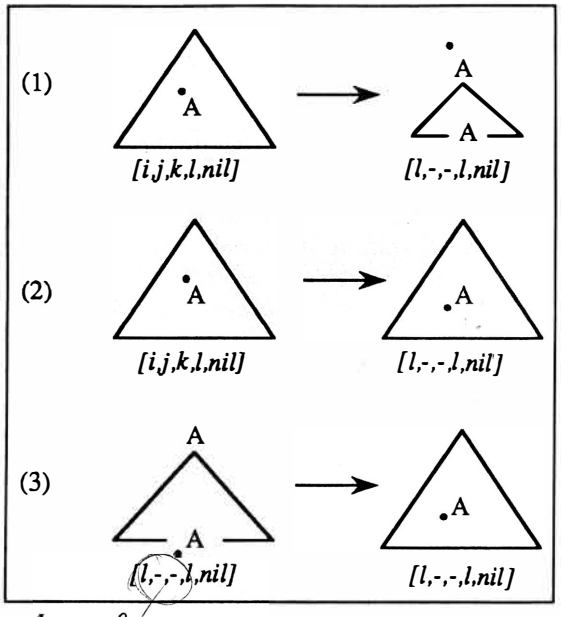


Figure 5: Predictor.

3.5.3 Completor

The *Completor* is a bottom-up processor that combines two states to form another state that spans a bigger portion of the input.

It consists of three possibly non-exclusive steps that apply when the dot is at position rb (right below). Step 1 considers that the next token comes from the part to the right of the foot node of an auxiliary tree adjoined on the dotted node. Steps 2 and 3 try to further recognize the same tree and concatenate boundaries of two states.

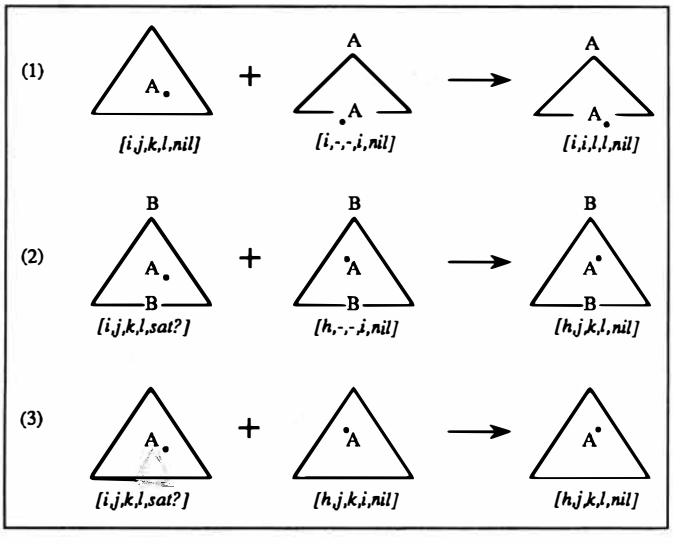


Figure 6: Completor.

Completor (see Figure 6):

- Step 1 combines a state of the form $s_1 = [\alpha, dot, rb, i, j, k, l, nil]$ such that $\alpha(dot) \in NT$, with a state $s_2 = [\beta, dot', lb, i, -, -, i, nil]$ such that $\beta \in Adjunct(\alpha, dot)$ and such that dot' is the address of the foot node of β. It adds the state $[\beta, dot', rb, i, i, l, l, nil]$ to $\mathcal{C}$.
- Step 2 combines a state of the form $s_1 = [\alpha, dot, rb, i, j, k, l, sat?]$ such that $\alpha(dot) \in NT$, and s.t. the node at address dot in α subsumes the foot node of α, with a state $s_2 = [\alpha, dot, la, h, -, -, i, nil]$. It adds the state $[\alpha, dot, ra, h, j, k, l, nil]$ to $\mathcal{C}$.
- Step 3 combines a state of the form $s_1 = [\alpha, dot, rb, i, j, k, l, sat?]$ such that $\alpha(dot) \in NT$ and s.t. the node at address dot in α does not subsume the foot node of α with a state $s_2 = [\alpha, dot, la, h, j, k, i, nil]$. It adds the state $[\alpha, dot, ra, h, j, k, l, nil]$ to $\mathcal{C}$.

3.5.4 Adjunctor

The *Adjunctor* is a bottom-up processor that combines two states by adjunction to form a state that spans a bigger portion of the input.

It consists of a single step.

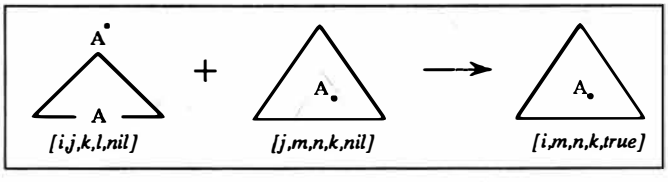


Figure 7: Adjunctor.

Adjunctor (see Figure 7):

It combines combines a state of the form $s_1 = [\beta, 0, rb, i, j, k, l, nil]$ and a state $s_2 = [\alpha, dot, rb, j, m, n, k, nil]$ such that $\beta \in Adjunct(\alpha, dot)$.

It adds the state $[\alpha, dot, rb, i, m, n, k, true]$ to $\mathcal{C}$.

3.6 An Example

We give an example that illustrates how the recognizer works. The grammar used for the example (see Figure 8) generates the language $L = \{a^n b^n e c^n d^n | n \geq 0\}$. The grammar consists of an initial tree α and an auxiliary tree β. There is a

null adjoining constraint on the root node and the foot node of β.

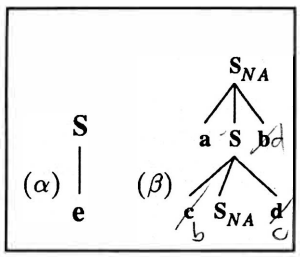


Figure 8: TAG generating $L = \{a^n b^n e c^n d^n | n \geq 0\}$

The input string given to the recognizer is: *aabbeccdd*. The corresponding chart is shown in Figure 9. For purpose of explanation, we have preceded each state with a number that uniquely identifies the state and we followed the state with the operation(s) that caused it to be placed into the chart. We used the following abbreviations: *init* for the initialization step, *pred(k)* for the Predictor applied to the state numbered by k, *sc(k)* for the Scanner applied to the state numbered by k, *compl(k+l)* for the combination with the Completor of the states numbered by k and l and *adj(k+l)* for the combination with the Adjunctor of the states numbered by k and l. With this convention, one can trace step by step the building of the chart. For example,

31. $[\beta, dot : 2, rb, 1, 4, 5, 8, t]$ *adj(30+24)*

stands for the state $[\beta, dot : 2, rb, 1, 4, 5, 8, t]$ uniquely identified by the number 31 which was placed into the chart by combining with the Adjunctor the states identified by the numbers 30 and 24.

The input is recognized since $[\alpha, 0, right, above, 0, -, -, 9, nil]$ is in the chart $\mathcal{C}$.

3.7 Implementation

The algorithm described in Section 3.5 can be implemented to follow an arbitrary search space strategy by using a priority function that ranks the states to be processed. The ranking function can also be defined to obtain a left to right behavior as in (Earley, 1968). Such a function may also very well be of statistical nature as for example in (Magerman and Marcus, 1991).

In order to bound the worst case complexity as stated in the next section, arrays must be used to implement efficiently the different processors. Due to the lack of space, we do not include the details of such implementation in this paper but they are found in (Schabes, 1991).

3.8 Correctness and Complexity

The algorithm is a general parser for TAGs with constraints on adjunction that takes in worst case $O(|G|^2 N n^6)$ time and $O(|G| N n^4)$ space, n being the length of the input string, $|G|$ the number of elementary trees in the grammar and N the maximum number of nodes in an elementary tree. The worst case complexity comes from the *Adjunctor* processor. An intuition of the validity of this result can be obtained by observing that that this processor (see Section 3.5.4) may be called at most $|G|^2 N n^6$ time since there are at most n^6 instances of the indices (i, j, k, l, m, n) and at most $|G|^2 N$ pairs of dotted trees to combine (α, β, dot). When it is used with unambiguous tree-adjoining grammars, the algorithm takes at most $O(|G|^2 N n^4)$-time[11] and linear time on a large class of grammars.

The proof of correctness consists in the proof of the invariant stated in Section 3.4.

Due to the lack of space, the details of the proofs of correctness and complexity are not given in this paper, but they are found in Schabes (1991).

3.9 The Parser

The algorithm that we described in section 3.5 is a recognizer. However, if we include pointers from a state to the other states (to a pair of states for the Completor and the Adjunctor or to a state for the Scanner and the Predictor) which caused it to be placed in the chart (in a similar manner to that shown in Figure 9), the recognizer can be modified to record all parse trees of the input string. The representation is similar to a shared forest. The worst case time complexity for the parser is the same as for the recognizer ($O(|G|^2 N n^6)$-time) but the worst case space complexity increases to $O(|G|^2 N n^6)$-space.

3.10 Extensions

The algorithm has been modified to handle extensions of TAGs such as substitution (Schabes et al., 1988), unification based TAGs (Vijay-Shanker and Joshi, 1988; Vijay-Shanker, 1991) and a version of multiple component TAG (see Schabes, 1990, for details on how to modify the parser to handle these extensions). It can also take advantage of lexicalized TAGs (Schabes and Joshi, 1989).

[11] This is a new upper-bound of the complexity of unambiguous TAG.

Input read	States in the chart	
	1. $[\alpha, dot: 0, la, 0, -, -, 0, nil]$ *init*	2. $[\beta, dot: 0, la, 0, -, -, 0, nil]$ *pred(1)*
	3. $[\alpha, dot: 1, la, 0, -, -, 0, nil]$ *pred(1)*	4. $[\beta, dot: 1, la, 0, -, -, 0, nil]$ *pred(2)*
a	5. $[\beta, dot: 2, la, 0, -, -, 1, nil]$ *sc(4)*	6. $[\beta, dot: 0, la, 1, -, -, 1, nil]$ *pred(5)*
a	7. $[\beta, dot: 2.1, la, 1, -, -, 1, nil]$ *pred(5)*	8. $[\beta, dot: 1, la, 1, -, -, 1, nil]$ *pred(6)*
aa	9. $[\beta, dot: 2, la, 1, -, -, 2, nil]$ *sc(8)*	10. $[\beta, dot: 0, la, 2, -, -, 2, nil]$ *pred(9)*
aa	11. $[\beta, dot: 2.1, la, 2, -, -, 2, nil]$ *pred(9)*	12. $[\beta, dot: 1, la, 2, -, -, 2, nil]$ *pred(10)*
aab	13. $[\beta, dot: 2.2, la, 2, -, -, 3, nil]$ *sc(11)*	14. $[\beta, dot: 2.2, lb, 3, -, -, 3, nil]$ *pred(13)*
aab	15. $[\beta, dot: 2.1, la, 3, -, -, 3, nil]$ *pred(14)*	16. $[\alpha, dot: 1, la, 3, -, -, 3, nil]$ *pred(14)*
aabb	17. $[\beta, dot: 2.2, la, 3, -, -, 4, nil]$ *sc(15)*	18. $[\beta, dot: 2.2, lb, 4, -, -, 4, nil]$ *pred(17)*
aabb	19. $[\beta, dot: 2.1, la, 4, -, -, 4, nil]$ *pred(18)*	20. $[\alpha, dot: 1, la, 4, -, -, 4, nil]$ *pred(18)*
aabbe	21. $[\alpha, dot: 0, rb, 4, -, -, 5, nil]$ *sc(20)*	22. $[\beta, dot: 2.2, rb, 4, 4, 5, 5, nil]$ *comp(21+18)*
aabbe	23. $[\beta, dot: 2.3, la, 3, 4, 5, 5, nil]$ *compl(22+15)*	
aabbec	24. $[\beta, dot: 2, rb, 3, 4, 5, 6, nil]$ *sc(23)*	25. $[\beta, dot: 2.2, rb, 3, 3, 6, 6, nil]$ *compl(24+14)*
aabbec	26. $[\beta, dot: 2.3, la, 2, 3, 6, 6, nil]$ *compl(25+13)*	
aabbecc	27. $[\beta, dot: 2, rb, 2, 3, 6, 7, nil]$ *sc(26)*	28. $[\beta, dot: 3, la, 1, 3, 6, 7, nil]$ *compl(26+9)*
aabbeccd	29. $[\beta, dot: 0, rb, 1, 3, 6, 8, nil]$ *sc(28)*	30. $[\beta, dot: 0, ra, 1, 3, 6, 8, nil]$ *compl(28+6)*
aabbeccd	31. $[\beta, dot: 2, rb, 1, 4, 5, 8, t]$ *adj(30+24)*	32. $[\beta, dot: 3, la, 0, 4, 5, 8, nil]$ *compl(31+5)*
aabbeccdd	33. $[\beta, dot: 0, rb, 0, 4, 5, 9, nil]$ *sc(32)*	34. $[\beta, dot: 0, ra, 0, 4, 5, 9, nil]$ *compl(33+2)*
aabbeccdd	35. $[\alpha, dot: 0, rb, 0, -, -, 9, t]$ *adj(34+21)*	36. $[\alpha, dot: 0, ra, 0, -, -, 9, nil]$ *compl(35+1)*

Figure 9: States constituting the chart for the input: $_0\,a\,_1\,a\,_2\,b\,_3\,b\,_4\,e\,_5\,c\,_6\,c\,_7\,d\,_8\,d\,_9$

4 Conclusion

We have shown that maintaining the valid prefix property for TAG parsing is costly because of the context-freeness of the path set of TAG derived trees.

In 1988, Schabes and Joshi introduced an Earley-style parser that satisfies the VPP however at a cost to its complexity ($O(n^9)$-time in the worst case but linear on some grammars). To our knowledge, it is the only known polynomial-time parser for TAG which satisfies the valid prefix property.

We have introduced a predictive left to right parser for TAGs which does not maintain the valid prefix property but takes at most $O(n^6)$-time in the worst case, $O(n^4)$-time for unambiguous grammars, and can behave linearly on some classes of grammars. The parser which we introduced is a practical parser since it often behaves better than its worst case complexity. It has been extended to handle extensions of TAGs such as unification based TAG and a restricted version of multiple component TAGs.

This predictive left to right parser can be adapted to other grammatical formalisms weakly equivalent to tree-adjoining languages (Joshi et al., Forthcoming 1990) such as linear index grammar, head grammars and a version of combinatory categorial grammars.

Bibliography

Jay C. Earley. 1968. *An Efficient Context-Free Parsing Algorithm.* Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

Jay C. Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

Saul Gorn. 1965. Explicit definitions and linguistic dominoes. In John Hart and Satoru Takasu, editors, *Systems and Computer Science.* University of Toronto Press, Toronto, Canada.

Karen Harbusch. 1990. An efficient parsing algorithm for Tree Adjoining Grammars. In 28^{th} *Meeting of the Association for Computational Linguistics (ACL'90)*, Pittsburgh.

Aravind K. Joshi, K. Vijay-Shanker, and David Weir. Forthcoming, 1990. The convergence of mildly context-sensitive grammatical formalisms. In Peter Sells, Stuart Shieber, and Tom Wasow, editors, *Foundational Issues in Natual Language Processing.* MIT Press, Cambridge MA.

Aravind K. Joshi. 1987. An Introduction to Tree Adjoining Grammars. In A. Manaster-Ramer, editor, *Mathematics of Language.* John Benjamins, Amsterdam.

T. Kasami. 1965. An efficient recognition and syntax algorithm for context-free languages. Technical Report AF-CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Bernard Lang. 1988. The systematic constructions of Earley parsers: Application to the production of $O(n^6)$ Earley parsers for Tree Adjoining Grammars. Unpublished manuscript, December 30.

M. David Magerman and Mitchell P. Marcus. 1991. Pearl, a probabilistic chart parser. In *Proceedings of the Second International Workshop on Parsing Technologies*, Cancun, Mexico, February.

Giorgio Satta and Alberto Lavelli. 1990. A head-driven bidirectional recognizer for lexicalized TAGs. Unpublished manuscript.

Yves Schabes and Aravind K. Joshi. 1988. An Earley-type parsing algorithm for Tree Adjoining Grammars. In 26^{th} *Meeting of the Association for Computational Linguistics (ACL'88)*, Buffalo, June.

Yves Schabes and Aravind K. Joshi. 1989. The relevance of lexicalization to parsing. In *Proceedings of the International Workshop on Parsing Technologies*, Pittsburgh, August. To also appear under the title *Parsing with Lexicalized Tree adjoining Grammar* in *Current Issues in Parsing Technologies*, MIT Press.

Yves Schabes and K. Vijay-Shanker. 1990. Deterministic left to right parsing of Tree Adjoining Languages. In 28^{th} *Meeting of the Association for Computational Linguistics (ACL'90)*, Pittsburgh.

Yves Schabes, Anne Abeillé, and Aravind K. Joshi. 1988. Parsing strategies with 'lexicalized' grammars: Application to tree adjoining grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics (COLING'88)*, Budapest, Hungary, August.

Yves Schabes. 1990. *Mathematical and Computational Aspects of Lexicalized Grammars.* Ph.D. thesis, University of Pennsylvania, Philadelphia, PA, August. Available as technical report (MS-CIS-90-48, LINC LAB179) from the Department of Computer Science.

Yves Schabes. 1991. A predictive left to right parser for tree-adjoining grammars. Technical report, Department of Computer and Information Science, University of Pennsylvania, Philadelphia. In preparation.

J. W. Thatcher. 1971. Characterizing derivations trees of context free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 5:365–396.

K. Vijay-Shanker and Aravind K. Joshi. 1985. Some computational properties of Tree Adjoining Grammars. In 23^{rd} *Meeting of the Association for Computational Linguistics*, pages 82–93, Chicago, Illinois, July.

K. Vijay-Shanker and Aravind K. Joshi. 1988. Feature structure based tree adjoining grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics (COLING'88)*, Budapest, August.

K. Vijay-Shanker and David J. Weir. 1990. Parsing constrained grammar formalisms. In preparation.

K. Vijay-Shanker. 1987. *A Study of Tree Adjoining Grammars.* Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

K. Vijay-Shanker. 1991. An unification based approach to Tree Adjoining Grammars. In preparation.

David J. Weir. 1988. *Characterizing Mildly Context-Sensitive Grammar Formalisms.* Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

D. H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.

February 13, 1991

Session B

Preprocessing and lexicon design for parsing technical text[1]

Robert P. Futrelle, Christopher E. Dunn, Debra S. Ellis and Maurice J. Pescitelli, Jr.

Biological Knowledge Laboratory
College of Computer Science 161CN
Northeastern University
360 Huntington Avenue
Boston, MA 02115

Internet: futrelle, chris, ellisds and mjp
all @corwin.ccs.northeastern.edu
Phone: (617) 437-2076 FAX: (617) 437-5121

ABSTRACT

Technical documents with complex structures and orthography present special difficulties for current parsing technology. These include technical notation such as subscripts, superscripts and numeric and algebraic expressions as well as Greek letters, italics, small capitals, brackets and punctuation marks. Structural elements such as references to figures, tables and bibliographic items also cause problems. We first hand-code documents in Standard Generalized Markup Language (SGML) to specify the document's logical structure (paragraphs, sentences, etc.) and capture significant orthography. Next, a regular expression analyzer produced by LEX is used to tokenize the SGML text. Then a token-based phrasal lexicon is used to identify the longest token sequences in the input that represent single lexical items. This lookup is efficient because limits on lookahead are precomputed for every item. After this, the Alvey Tools parser with specialized subgrammars is used to discover items such as floating-point numbers. The product of these preprocessing stages is a text that is acceptable to a full natural language parser. This work is directed towards automating the building of knowledge bases from research articles in the field of bacterial chemotaxis, but the techniques should be of wide applicability.

1. INTRODUCTION

The Biological Knowledge Laboratory focuses on the analysis of research articles in the field of bacterial chemotaxis (Futrelle, 1989, 1990b). We are building a corpus consisting of the 1000 or so articles that make up the published record of the field since its inception in 1965. As the corpus is built we are attempting to use syntactic and semantic analysis to convert the corpus to a knowledge base. But the texts are complex -- they have a superstructure that includes title, authors, abstract, sections, paragraphs, bibliography, etc. They also contain sub- and superscripts, italics, Greek letters, formulas, and references to figures, tables, and bibliographic items. Another major component of technical documents that has been ignored is graphics, which requires its own analysis; we have a separate project devoted to graphical analysis and understanding (Futrelle, 1990a).

[1] This work was supported by the Division of Instrumentation and Resources of the National Science Foundation, grant number DIR-88-14522.

In this paper we describe procedures we have implemented and resources we have developed for preprocessing these complex documents. The preprocessing produces text which retains all important details of the original but is in a form that a conventional natural language parser can use without major modifications.

The preprocessing software runs in part under Unix (for LEX) and in part under Symbolics Genera 8.0 using their *Statice* database system for the lexicon. The Alvey Natural Language Toolkit (Briscoe, et al, 1987) is used for the subgrammar analysis. We have used Alvey on the Symbolics, Suns and on Mac II's. The systems described here are sentence-oriented, leaving to other software the task of organizing the structures above the sentence level.

Most research on natural language processing is restricted to text which does not contain complex orthography or has had it stripped away. This has prevented the application of computational linguistics to most technical documents and technical documents are a huge and important repository of knowledge. Though our contribution is primarily a technical one, it is one that is sorely needed if progress is to be made.

2. THE PROBLEMS AND THEIR SOLUTION

To appreciate the type of problems that arise in text analysis, consider the various uses of a punctuation mark, the period. In the sentence, "Bacteria swim." the item "swim." that includes the period is not a word, it is the word "swim" followed by end-sentence punctuation. On the other hand, the period in "etc." is not (necessarily) a sentence end marker. The period in "7.3", however, is an integral part of the number. The comma is normally used to mark phrases and clauses, but it is used as an integral part of the number "32,768" or the chemical name "2,6-diaminohexanoic acid" (the essential amino acid, lysine). Superscripts can play the role of an isotopic indicator, "^{3}H" for tritium, or a footnote[2].

[2] ...or a bibliographic reference, as in, "Smith found this effect earlier[7]."

We have found a way to deal with all of these problems. The documents are first encoded (marked up) as they are entered by a trained editor/typist using an editor which supports the Standard Generalized Markup Language (SGML) (Bryan, 1988; van Herwijnen, 1990). The complex items in the marked-up text are then broken into their constituent tokens and selectively reassembled so that every token or contiguous sequence of tokens is *resolved* in some way. The resolution of a token sequence is done by first looking for the sequence in a phrasal lexicon. If found, the sequence is replaced by its lexical item. If a token sequence is not in the lexicon, an attempt is made to parse it using specialized subgrammars. If this fails, the item is flagged for analysis by a human editor or lexicographer to see if it is an error or a new lexical item.

The word "salt" is a single token entry in the lexicon. The sequence, "sodium chloride" is a two token entry. The item "CO_2" which is represented by seven tokens is found as a single item in the lexicon. But it is not appropriate to represent most numbers in the lexicon, because they form an essentially *unbounded class*[3]. For example, the number "3.4×10^{-8} " (made up of 17 tokens) is not in the lexicon. It is analyzed by a subgrammar and found to be a legally formed number in scientific notation. The number is replaced by a structure which includes the lexical item "num", a noun which the natural language parser can deal with. After preprocessing, the text is passed on to a full natural language parser for syntactic and semantic (logical form) analysis. Currently, we use the GPSG-based parser from the Alvey toolkit for both subgrammar analysis and full natural language parsing (Briscoe, et al, 1987; Ritchie, et al, 1987).

3. THE PROCESSING SEQUENCE

The processing sequence is outlined in Figure 1. Each stage can produce a file as output that can be the input to the next stage, so the analyses do not have to be synchronous. The *pre*processing stages are stages 1-6.

[3] Certain numbers such as cell strain designators or the familiar "Boeing 747" would be in the lexicon.

Stage 0: Obtain selected articles from primary biological literature, 1960-1990
Form 0: *word complex-orthographic-item word word floating-point-number punctuation ....*
Stage 1: SGML encoding (tagging) while typing in article using SGML-based editor
Form 1: *sentence-start-tag word tagged-complex-item word word tagged-number ....*
Stage 2: Tokenization using regular-expression analyzer generated by LEX
Form 2: *SGML-symbol string complex-item-token ... tokens-for-number SGML-symbol ....*
Stage 3: Lexicon lookup in token-based phrasal lexicon
Form 3: *found-item found-item found-item not-founds found-item not-founds ....*
Stage 4: Subgrammar analysis using Alvey syntactic and semantic tools
Form 4: *found-item found-item found-item analyzed-structure not-found*
Stage 5: Editor and lexicographer at the workbench resolve any remaining unknowns
Form 5: *found-item found-item found-item analyzed-structure added-to-lexicon*
Stage 6: Natural language parsing using Alvey GPSG-based tools
Form 6: *Parse trees and logical form structures*
Stage 7: Building knowledge frames . . . .

Figure 1. Schematic view of the successive stages of corpus processing. "Form *n*" lists typical items in the stream of text which result from the processing in Stage *n* and are the input to Stage *n+1*. There is not an absolutely tight correspondence between the items in successive forms in this figure, due to the complexity of the analysis. The underlined stages denote the preprocessing stages which are currently implemented and explained in some detail in this paper.

STAGE 0: Obtaining Selected Articles – In many cases, these articles are only available in bound journals. The originals are scanned for diagram entry, but the typing (with simultaneous markup) is done from photocopies when necessary.

STAGE 1: SGML Markup – Markup languages such as SGML allow us to add *markup* to a text of a document to specify its logical structure. Thus, in SGML, one would specify, using *tags,* that certain words formed a *section heading* without committing to stylistic details such as font, font size, or the positioning of the heading with respect to the margin. For example, the text that begins the subsection you are reading would be encoded in SGML as:

(1) <SS1><ST> STAGE 1: SGML Markup </ST> <P><U.S>Markup languages such as SGML add <E1>markup</E1> to a text of a document to specify its structure. </U.S>

In (1) the SGML *tags* enclosed by braces have the following meanings:

(1a)	<SS1>	=	subsection start-tag
	<ST>	=	section title start-tag
	</ST>	=	end of section title
	<P>	=	paragraph start tag
	<E1>	=	emphasis start tag
	</E1>	=	emphasis end
	<U.S>	=	sentence start tag
	</U.S>	=	sentence end tag

The SGML encoding of (1) is, in turn,

(2) <SS1><ST>STAGE1: SGML Markup – ... </U.S>

which shows that we can satisfy *Becker's Criterion* (Becker, 1975) that states that any technique that claims to be useful and generally applicable should be able to analyze the very text which explains the technique!

In (2) items such as "<" are SGML *entities*; this one denoting the reserved character, "<" (less than). The tags used here are drawn

from the American Association of Publishers' (AAP) set, the Electronic Manuscript Standard (EMS), with the addition of our own user-defined tags such as the sentence tags, <U.S> and </U.S>. SGML is an ISO standard (#8879). SGML specifies a system in which tags and entities can be defined and used so that an arbitrarily complex text can be translated to a standard form which uses only the ASCII character set so it can be disseminated widely and dealt with uniformly by a variety of systems.

The encoding (markup) of the text is done using an SGML editor that makes the process efficient and checks that the text complies with our SGML syntax specifications, e.g., no sentence-start tag can be entered until the previous sentence-end tag has been entered. The particular system we use is Author/Editor (Softquad, Toronto, Canada) running on Mac II's .

The example sentence – Here is the example sentence we will use to illustrate our preprocessing strategy. It is first presented as it might appear in a research article source, but laid out for easy comparison with the SGML form which follows:

(3a) Cells were suspended in medium containing

(3b) 3.05×10^{-2} μM

(3c) L-[*methyl* -^{3}H]-methionine,

(3d) α-methylaspartate

(3e) and AIBU[8].

Here is the SGML encoding of the example sentence:

(4a) <U.S>Cells were suspended in medium containing

(4b) 3.05×10^{−2}µM

(4c) <SCP>L</SCP>-[<IT>methyl</IT>-³H]-methionine,

(4d) <GK>a</GK>-methylaspartate

(4e) and AIBU <RB>8</RB>.</U.S>

The "µ" entity stands for the Greek letter mu. "<SCP>" indicates small caps, "<IT>" indicates italics and "<RB>" is a bibliographic reference tag. Note that small caps and italics are encoded because they are standard typographical conventions used in chemical names; otherwise the *appearance* of items is not encoded.

STAGE 2: Tokenization – We use an analyzer generated by LEX (Aho, Sethi and Ullman 1986) to tokenize the input. It uses a regular expression grammar to identify the primitive elements of the SGML encoded text. The six classes of tokens produced by this stage are shown in Table 1. Note that "token" as we use it here includes a parenthesized pair (for numbers), not just a contiguous sequence of non-blank characters.

Table 1. The input and output forms for the tokenization stage, Stage 2.

Input Class	Output Format	Example Output
ASCII text strings	*string*	"Cells"
numbers	(num *string*)	(num "05")
special characters	(*string*)	(".") (",") ("(")
SGML tag	*symbol*	<U.S>
SGML entity	*symbol*	|µ|
no-white-space	nws	nws

For each class, the original ASCII representation has been preserved, either by including the string itself or using a Lisp symbol whose print representation is the ASCII representation. As an example, the outputs from tokenizing (4a) and (4b) are the 7 token sequence (5a) and the 20 token sequence (5b):

(5a)<U.S> "Cells" "were" "suspended" "in" "medium" "containing"

(5b) (num "3") nws (".") nws (num "05") nws |×| nws (num "10") nws ^{nws |−| nws (num "2") nws} |µ| nws "M"

The white spaces in the original text have been complemented to yield the *nws* symbol to indicate that the tokenized elements were originally abutted. This is necessary for disambiguation of complex sequences, and it makes normal prose easier to read at this stage.

Stage 3: Lexicon Lookup – At this point, a lexicon is consulted for each sequence of tokens contained in a title, section heading, sentence, etc. For our example, the token sequence generated from the full sentence (4) is handed to the lexicon lookup routine as the 73 token list,

(6) ("Cells" "were" "suspended" ... <GK> nws "a" nws </GK> nws ("-") nws "methylaspartate" ... (num "8") nws </RB> nws ("."))

(notice our ellipsis). The lexicon lookup stage attempts to match sequences of tokens from the input to items found in the lexicon. The lexicon is an extended phrasal lexicon, in which each lexical entry is a sequence of one or more tokens. Five typical lexical items are

"cells"

"sodium chloride"

"<GK>a</GK>-methylaspartate"

"<GK>"

"."

Note that in the lexicon, the *nws* (no-white-space) tokens are removed by concatenation for both storage and lookup. A lexical item L (one or more tokens) is a *prefix* if there are longer items in the lexicon (more tokens) with the same initial items as L. The first token of all items in the lexicon is listed as a separate entry. But some of these and some multiple token entries never function as independent *stand-alone* items and are noted as such in the lexicon. For example the SGML tag tokens "<GK>" and "<IT>"indicating that Greek and italicized characters follow never function as separate items.

To efficiently and reliably find multi-token items, certain information is precomputed and stored in the lexicon. For example, the items "sodium", "chloride", "sodium chloride", "sodium bromide" "sodium iodide" might all appear in the lexicon. When "sodium chloride" appears in the source text, it is that two-item entry that we want identified, not the two separate words. To assure that this happens the prefix list ((3 2)) is computed and attached to "sodium". This says that there are 3 items of length 2 that begin with "sodium", so the next item in the source, "chloride" is attached and the two-word item is found and returned by the lexicon lookup. Prefix lists can be complex, forming trees rooted at the initial item. The prefix lists prevent the search for a single item from continuing to the end of the sentence, because they put explicit bounds on the lengths of all items that could possibly match, given any prefix.

The output from the lexicon lookup stage for (6) is the list

(7a) ("Cells" "were" "suspended" "in" "medium" "containing"

(7b) (?? ((num "3") nws))

"."

(?? (nws (num "05") nws |×| nws (num "10") nws ^{nws |−| nws (num "2") nws}))

"µM"

(7c) "<SCP>L</SCP>-[<IT>methyl</IT>-³H]-methionine" ","

(7d) "<GK>a</GK>-methylaspartate"

(7e) "and" "AIBU"

(?? (<RB> nws (num "8") nws </RB> nws))

".")

There are three unknown item sequences here, shown broken out in (7b) and (7e) as (??....) forms. The first two are parts of the number 3.05×10^{-2}. The third is a bibliographic reference. The "." in the number in (7b) and the "." at the end of the sentence in (7e) are recognized since "." is a stand-alone item. <SUP> it is a prefix for entries such as "³H-ethanol" but it is not stand-alone, so it is included in the unknown in (7b). Note that the strings which are the lexicon identifiers for complex items such as the chemical name in (7d) retain their original SGML markup, without the no-white-space symbols introduced by tokenization. In an interactive system, these items could be presented on a screen by interpreting the markup according to a style specification and producing the indicated orthography, e.g., α-methylaspartate.

Stage 4: Subgrammar analysis – The reason that the three unknown items were unrecognized in the previous step is that they were parts of lexical items that belong to two of the unbounded classes of lexemes. The job of the subgrammar is to analyze this type of unknown which can include numbers, number ranges, simple ratios, references and page numbers. Each class has an associated structure for representing its instances. In our previous example we had two unknown token sequences and one lexical item, which when taken together correspond to the number 3.05×10^{-2}:

(8) (?? ((num "3") nws))

"."

(?? (nws (num "05") nws |×| nws (num "10") nws ^{nws |−| nws (num "2") nws}))

We have written a context-free grammar to recognize this token stream as a number in scientific notation and place a structure in the output stream of the general form

(9) ("num" *SGML-string Lisp-num-form*)

For our example (8) this would result in:

(10) ("num" "3.05×10^{−2}" 3.05E-2)

The number structure consists of three fields. The first, "num", is a lexical item, the noun which represents all numbers. The parser for doing the later syntactic analysis of this sentence will access the feature-value list associated this noun. The second field contains the SGML encoding of the number. This can be used for displaying the number on the screen. The third field contains a Lisp-readable form of the number.

Another structure recognized by subgrammar analysis is the bibliographic reference, (7e). The structure produced by the analysis has the form:

(11) ("$bibref$" *SGML-string List-of-contents*)

When the token sequence from (7e) is recursively analyzed, the result is

(12) ("$bibref$" "<RB>8</RB>" (("num" "8" 8)))

In this example, the bibliographic reference structure contains a number structure. In general, any sequence of lexical items, structures and unrecognized token streams

can be placed in the *List-of-contents* for bibliographic references.

Subgrammar analysis of expressions such as (8) involves first creating a stream without the "??" tokens and without the actual integers ("3", "05", "10" and "2") and with the "ordinary" words replaced by simple placeholders, e.g., "$word$". Critical elements such as nws, <SUP> |−|, etc. are retained.

Once this simplified stream is available, the parse is done according to the subgrammar specialized for numbers, bibliographic references, etc. But the output of the subgrammar analysis must produce a new stream which includes forms such as in (10) and (12) as well as all of the original words. To do this we take advantage of the compositional semantics built into the Alvey parser. The semantic attachment facilities in Alvey allow references to daughter nodes by number and the inclusion of simple lambda forms. But in addition, arbitrary lisp forms can be included. We define semantic rules with lisp forms included. The Alvey semantics then works compositionally by walking up the parse tree. This allows the semantic interpretation to generate the Common Lisp source code for a translator of the original stream, e.g., of (7a-e). When this translator is applied to the original stream, all "??" items which parse are replaced by forms such as (10) and (12) and all words such as "Cells" "were", etc. are simply copied to the output. All "??" items that remain are either ill-formed or are items not yet in the lexicon. Note that a separate translator is built for each sentence. But the construction is simple and deterministic and therefore rapid. Lisp's ability to treat code as data is what we're exploiting here.

The syntactic role of some of the special forms found by the subgrammar is subtle. Thus, in

(13) "This was discovered by Smith when he was working at the MBL[19]."

the bibliographic reference does not act like any familiar syntactic constituent. But in the following form the reference functions as a noun,

(14) "Commonsense knowledge is discussed in (Davis, 1990)."

In the full natural language parsing (Stage 6) there will be additional categories and grammar rules to allow such structures to be treated properly.

When the translator generated by the semantic interpretation of the subgrammar parse is applied to (7a-e), the final form which results is

```
(15a)  ( "Cells" "were" "suspended""in"
         "medium" "containing"

(15b)  ("$num$"
          "3.05&times;10<SUP>minus;2
          </SUP>" 3.05E-2)
        "&micro;M"

(15c)  "<SCP>L</SCP>-[<IT>methyl</IT>-
        <SUP>3</SUP>H]-methionine" ","

(15d)  "<GK>a</GK>-methylaspartate"

(15e)  "and" "AIBU"
         ("$bibref$" "<RB>8</RB>"
            (("$num$"  "8"  8))) "." )
```

This preserves all of the details of the original text. Every form is an item or contains an item that can be found in the lexicon and one that will allow a proper screen display (cf. (16) below). Lisp forms of numbers and citation information are also available.

The subgrammars are simple and deterministic so the parses are fast compared to the later full natural language parses.

Stage 5: The Lexicographer's Workbench

Natural language parsing cannot be done until all items are resolved by the lexicon, so unknown items are passed on to the editor and the lexicographer (humans). Errors in the original source and errors in our own re-entry can be caught at this stage. What remain are items that need to be added to the lexicon. These additions are made using the Lexicographer's Workbench which is currently under development. In the Workbench a collection of analytical tools and heuristic procedures are used to tentatively classify new items which are then presented to the lexicographer for simple approval or more rarely for special treatment. Morphological

analysis is useful, e.g., certain classes of enzyme names have the suffixes "tase" or "ase" as in "phosphatase" or "nuclease". This means that new words can be analyzed and suggestions made as to their classification. Alvey has a sophisticated morphological analysis package which we are experimenting with in which the rules are user definable (Ritchie, et al 1987).

One difficult task is the identification of new *phrasal* items, a difficulty emphasized by Amsler (Amsler, 1989). For example, consider the case in which "sodium", "chloride", "bromide" and "sodium chloride" are in the lexicon but "sodium bromide" is not. If "sodium bromide" appeared in the input it would not even be flagged as an unknown. Nevertheless, we would want the Workbench to be provided with the heuristic that chemical name sequences are most likely chemical names themselves. Thus the workbench would make the decision itself and insert "sodium bromide" in the lexicon with the proper feature/value specs. This decision would, as all others, be subject to review by the lexicographer or application field specialist.

Stage 6: Natural language parsing – When the lexical items are extracted from (15), the result is

(16a) ("Cells" "were" ""suspended" "in" "medium" "containing"

(16b) "num" "µM"

(16c) "<SCP>L</SCP>-[<IT>methyl</IT>-³H]-methionine" ","

(16d) "<GK>a</GK>-methylaspartate"

(16e) "and" "AIBU" "$bibref$" ".")

This is the input to the natural language parser. The grammar furnished with the Alvey tools is large and covers a wide variety of constructions. Nevertheless, it will take further extensions to get acceptable coverage of the scientific prose in our corpus. This is work in progress. A semantics for this large grammar is under development (C. Grover, personal communication). In addition, a more efficient, LR(1) parser is being built to improve the performance over the chart parser currently available in the Alvey Toolkit (J. Carroll, personal communication).

Stage 7: Building Knowledge Frames – We have studied papers in our corpus in an effort to identify all of the major semantic constructions. One type deals with the experimental details themselves such as the techniques used and the results seen. The other deals with scientific argumentation – how models are used to suggest experiments and how results reinforce or weaken various hypotheses that might explain them. Our goal is to design knowledge frames for the different semantic structures we have found. Then the logical forms produced by parsing would be used as input to a system which generates instances of the appropriate knowledge frames representing the sentences. (This is also work in progress.) Furthermore, these knowledge frames can be connected together into superstructures representing coherent arguments for or against a given proposition. Taken together, these frame instances and their connecting frames compose the *knowledge base* which would underlie our "Scientist's Assistant" system, a system for answering both general and specific queries about the contents and arguments that are to be found in our corpus.

4. DISCUSSION

Because of the complexities of technical text notation and the availability of a comprehensive standard, we decided to use SGML for text markup. Then we designed a token-based phrasal lexicon for resolving the complex items generated by the markup. This lexicon is robust because it handles everything from simple words to complex multi-word chemical names containing Greek letters, commas, superscripts and more. In addition, our subgrammar analysis handles unbounded class items that cannot be accommodated in the lexicon such as numbers in scientific notation and bibliographic references.

The work closest to ours is the preprocessing done for the LOB corpus (Booth, 1987). Unfortunately, the SGML standard was not available to that project at the time, so they had to invent their own orthographic coding

schemes and a pre-editing phase similar to ours to break the text into taggable units. There are many differences between the projects. One of these is in the design of the lexicon. The LOB group decided to develop a compact lexicon which includes only the base forms. Possessives or contracted forms such as "Smith's" or "it's" are not included. Because secondary storage is rapidly becoming less expensive and because modern database and file structure designs allow very rapid access to large lexicons we have opted for a very "flat" lexicon in which *every* variant form encountered in the corpus is stored as a separate entry. This includes capitalized words appearing at the beginning of sentences, etc. We add the variants of the base forms to the lexicon *only* when they are found in our corpus. Our own statistical analysis of large corpora such as the Brown Corpus show that the inclusion of these variant forms will probably add no more than 50% to the lexicon size over a lexicon that has only the base forms.

If we had only included base forms then other difficulties would crop up in attempting to map between found entities and the base forms. We avoid these difficulties by including the variant forms and flagging them to indicate their usage restrictions. We would flag "There" as a form only expected as a sentence initial (and fully equivalent to "there") whereas "DNA" would only be expected in fully capitalized form.

Another major activity in text encoding is the Text Encoding Initiative or TEI (Sperberg-McQueen and Burnard, 1990). They have been focusing on text in the humanities so they have been concerned with a different set of problems such as encoding verse, stage directions, foreign language quotations, etc. Neither the TEI not the LOB groups seemed to have directly faced the issues of how to interface the marked up text with the available parsing technology as we have.

SGML allows the user to design their own set of tags, entities and rules so we had to make some design decisions. Our design is constructed pragmatically to make it usable by an editor/typist who is not a scientist. For instance, we have used a special tag "<RB>" for a bibliographic reference which might be represented by a superscript or by the conventional "(Shepard, 1978)". And we have opted to use the simple superscript tag "<SUP>" for both algebraic exponents as in "$3.05x10^{-2}$" and isotope indicators as in "^{3}H". The subgrammar and lexicon lookup, respectively, resolve these latter two items. This allows the typist to encode source text primarily on the basis of its appearance, rather than its semantic (scientific) content.

We are constantly asked why we do not use OCR techniques (optical character recognition) or go directly to publishers for electronic versions of the papers in our corpus. Again, these are pragmatic decisions, peculiar to this point in time. Because OCR error rates are still relatively high, especially for technical text, and because OCR systems do little or no markup, we can produce accurate transcriptions and markup more cost-effectively by having a skilled typist/editor rekey the text. Most of our corpus (covering 30 years) does not exist anywhere in electronic form, and the wide variety of proprietary schemes used by printing firms for electronic typesetting is a nightmare to untangle.

In the future, technical word processing systems will be developed that will allow scientist authors to enter their text with the proper logical tagging but without the system obtruding on their work. The systems we are developing will be able to take advantage of such electronic documents as they become available.

Many authors have argued cogently and at length that multi-word items, idioms, punctuation and other complexities of real text require a comprehensive approach (Becker, 1975; Besemer and Jacobs, 1987; Amsler, 1989; Nunberg, 1988, 1990). The methods described here can serve as a foundation for any comprehensive system that must deal with the lexical, syntactic and semantic aspects of real-world technical text.

ACKNOWLEDGEMENTS

We thank John Carroll and Claire Grover for discussions of the Alvey tools, including the semantic component.

REFERENCES

Aho, A.; Sethi, R. and Ullman, J. 1986. Compilers; Principles, Techniques, and Tools. Addison-Wesley Publishing Company, Inc., Reading, MA.

Amsler, Robert A. 1989. Research Toward the Development of a Lexical Knowledge Base for Natural Language Processing. *Proceedings of the Twelfth Annual International ACMSIGIR Conference on Research and Development in Information Retrieval.* Cambridge, MA : 242-249.

Becker, J.D. 1975. The Phrasal Lexicon. In *Proceedings Interdisciplinary Workshop on Theoretical Issues in Natural Language Processing.* Cambridge MA : 70 - 73.

Besemer, David J. and Jacobs, Paul S. 1987. FLUSH: A Flexible Lexicon Design. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics..* Stanford University, Stanford, CA : 186 - 192.

Booth, Barbara 1987. Text input and pre-processing: Dealing with orthographic form of texts. In *The Computational Analysis of English. A Corpus-Based Approach.* (Longman, London).

Briscoe, E.; Grover, C.; Boguraev, B. and Carroll, J. 1987 A Formalism and Environment for the Development of a Large Grammar of English. *Proceedings of the 10th International Joint Conference on Artificial Intelligence,.* Milan, Italy: 703-708.

Bryan, M. 1988. *SGML: An Author's Guide to the Standard Generalized Markup Language.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts.

Futrelle, R. P. 1989. *An Introduction to the Biological Knowledge Laboratory.* Technical Report NU-CCS-89-15. College of Computer Science, Northeastern University.

Futrelle, R.P. 1990a. Strategies for Diagram Understanding Object/Spatial Data Structures, Animate Vision, and Generalized Equivalence. *Proceedings of the 10th International Conference on Pattern Recognition.* Atlantic City, NJ: 403-408.

Futrelle, R. P. 1990b *Current Activities in the Biological Knowledge Laboratory (BKL).* Technical Report NU-CCS-90-20. College of Computer Science, Northeastern University.

Nunberg, Goeffrey 1988, 1990. *The Linguistics of Punctuation.* Technical Report P88-00142. XEROX System Sciences Laboratory, Palo Alto Research Center, Pal Alto, CA (U. Chicago Press, 1990, to appear).

Ritchie, Graeme D.; Pulman, Stephen G.; Black, Alan W. and Russell, Graham J. 1987. A Computational Framework for Lexical Description. *Computational Linguistics,* Vol. 13, No. 3-4: 290-307.

Sperberg-McQueen, C. M. and Burnard, Lou, editors. 1990. *Guidelines For the Encoding and Interchange of Machine-Readable Texts.* Document Number: TEI P1. Draft: Version 1.0. The Association for Computers and the Humanities; The Association for Computational Linguistics; The Association for Literary and Linguistic Computing.

van Herwijnen, Eric 1990. *Practical SGML.* Kluwer Academic Publishers, Dordrecht, The Netherlands.

INCREMENTAL LL(1) PARSING IN LANGUAGE-BASED EDITORS

John J. Shilling
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
shilling@cc.gatech.edu

ABSTRACT

This paper introduces an efficient incremental LL(1) parsing algorithm for use in language-based editors that use the structure recognition approach. It features very fine grained analysis and a unique approach to parse control and error recovery. It also presents incomplete LL(1) grammars as a way of dealing with the complexity of full language grammars and as a mechanism for providing structured editor support for task languages that are only partially structured. The semantics of incomplete grammars are presented and it is shown how incomplete LL(1) grammars can be transformed into complete LL(1) grammars. The algorithms presented have been implemented in the fred language-based editor

INTRODUCTION

This paper introduces an efficient incremental LL(1) parsing algorithm for use in language-based editors that use the structure recognition approach. It is motivated by a style of interaction that parses the user input at intervals of very small granularity. A second motivation for the algorithm is the problem of changes internal to the editing buffer. Because incremental analysis can occur after each keystroke, an unrestricted parser will attempt to include too much into its focus before a change is complete causing the editor to detect erroneous states that will become irrelevant as the user completes the change. The parsing algorithms presented in this paper use the user focus as a guide in restricting parsing. The algorithm presented has been implemented in the fred language-based editor [Shi83, Shi85].

Incomplete LL(1) grammars are introduced as a way of dealing with the complexity of full language grammars and as a mechanism for providing structured editor support for task languages that are only partially structured. Incomplete grammars were introduced by Orailoglu [Ora83] for the fred editor [Shi85, Shi86] as a method of dealing with the complexity of full language grammars. Incomplete grammars allow incremental refinement of language grammars and also allow grammars to be defined for languages that are not LL(1). Defining an incomplete grammar for a non-LL(1) language allows the editor to give structured support for the LL(1) subset of the language rather than disallowing the language completely. Another useful application of incomplete grammars is in providing structured support for tasks whose languages are only partially structured. An example of this is a grammar that facilitates structured support for editing LaTeX documents. A LaTeX documents contains structured elements but much of the document can be treated as unstructured text.

This paper introduces incomplete LL(1) grammars and characterizes their parsing semantics. It then shows how the grammars can be translated into conventional LL(1) grammars, eliminating the need for specialized parsing algorithms.

INCREMENTAL LL(1) PARSING

The goal of incremental parsing is to re-establish a correct structuralization of the user's editing buffer after changes have been made. The approach taken must differ from straightforward *once-only* top-down parsing because a once-only parser never needs to reverse decisions after they are made. In incremental parsing decisions are unmade and sections of the parse tree are deleted, transformed, and grafted into new locations. At the same time, the amount of parsing actually done must be limited if the algorithms are going to provide real-time response to a user. The algorithms must first establish the scope of modifications and efficiently restructure the parse tree within this scope.

The parsing method described in this paper is more fine grained than previous methods. The goal is to restructure the editing buffer after each text-modifying keystroke of a user. The challenge is that it is often not possible to achieve a complete, correct structuralization because the user is in the process of making a change that is not yet complete. On the other hand, the user

```
while (TRUE)
    <user change>
    <retokenization>
    <preparation of Parse Tree (Sweep)>
    <incremental parse>
    <semantic update>
```

Figure 1: Change-Update Loop

should be notified at the earliest possible moment if an error is made. The solution to this conflict is to implement what is called *follow-the-cursor* parsing with *soft templates*. As a user makes changes the method will parse only up to (and including) the token that contains the cursor. This keeps it from trying to parse past the cursor when a user has not yet completed a change. Unsatisfied elements of a production are indicated to the user as soft templates. Soft templates visually show the user what is missing in the parse tree. They are *templates* in that they should a valid production at the point they appear but they are *soft* because they do not constrain the user in any way. Further text is brought into consideration through cursor movement. The incremental LL(1) parsing algorithms presented here are a generalization of the table driven LL(1) parsing algorithms presented by Lewis, Rosenkrantz, and Sterns [PLRS76] and use *Select*, *Nullable* and *Follows* tables.

THE CHANGE-UPDATE LOOP

As a user changes a program the editor executes the loop illustrated in figure 1 to achieve a correct restructuralization. The localized region of change must be retokenized, the tree prepared, and the new tree state incrementally parsed. The data structures of the non-incremental algorithm are extended to facilitate incremental parsing. The parsing queue is modified to handle both tokens and non-terminals so that subtrees from the parse tree do not always have to be broken down into tokens as they are moved to the parse queue. This means that the parsing tables must be expanded to take account of non-terminals. We now assume that both the *Select* table and the *Follows* table cross reference non-terminals with both tokens and non-terminals.

TOKENIZATION

We will regard the tokenization phase as a black box process that produces a series of tokens from the localized region of change. It is assumed that incremental tokenization produces a queue of tokens and two markers in the parse tree denoted the Lexical Left Boundary and the Lexical Right Boundary. These markers point out the region along the frontier of the parse tree (inclusive) that has become invalid as a result of the new tokenization.

TREE PREPARATION - SWEEP

The next step in the change-update loop is the tree preparation process called Sweep. This is the process that breaks down the affected region of the parse tree and prepares the tree for the parsing algorithm. Two nodes of the parse tree have special meaning in this process. They are called the *Common Ancestor* and the *Royal Node* and are defined as follows:

- The **Common Ancestor** is the lowest node in the parse tree that is an ancestor of both the Lexical Left Boundary and the Lexical Right Boundary.
- The **Royal Node** is the highest node in the parse tree such that the Lexical Left Boundary is the first token of the production[1]. If there is no such node then the Royal Node is the Lexical Left Boundary.

Two basic ideas drive the tree preparation. The first is that the region of the tree defined by Lexical Left Boundary, Lexical Right Boundary and Common Ancestor is invalidated because the tokens along its frontier have been recalculated. The second is that the subtree of the parse tree rooted at Royal Node is suspect because it was instantiated on the basis of a token that has been altered.

Figure 2 shows the Sweep algorithm. It begins by identifying the Common Ancestor and the Royal Node and then cleans the region modified by the lexical tokenization. This is a wedge in the parse tree that is bounded by the path from the Lexical Left Boundary to the Common Ancestor to the Lexical Right Boundary. All nodes on the interior of the modified region are deleted except the direct sons of the nodes along the boundary.

The algorithm must now decide what to do about the Royal Node. We distinguish two cases in dealing with the Royal Node based on the relationship between the Royal Node and the Common Ancestor. If the Royal Node is a descendent of the Common Ancestor then there is no conflict because there are no tokens in the subtree rooted at Royal Node. If Royal Node is the same as, or an ancestor of the Common Ancestor then the subtree rooted at the leftmost son of Common Ancestor is clipped. This will in general leave parts of the parse tree intact that may not be valid with the new tokenization.

Before exiting, the the Sweep algorithm pushes the current parse pointer back to the left in the parse tree

[1] We will ignore non-significant nodes such as error nodes and (usually) white space in this presentation

Sweep(LexLeftBound, LexRightBound):

```
CommonAncestor = CommonAncestor(LexLeftBound, LexRightBound);
RoyalNode = RoyalNode(LexLeftBound);

CleanRegion(LexLeftBound, LexRightBound, CommonAncestor);

if (RoyalNode in subtree of CommonAncestor)
    DeleteSubtree(RoyalNode);
else
    DeleteSubtree(LeftmostSon(CommonAncestor));
endif

BackUp(Parse Position);
```

Figure 2: Sweep

as far as it can until it hits a token. The first non-terminal to the right of that token becomes the location of the current parsing position.

INCREMENTAL PARSING

We now enter the actual incremental parsing algorithm. The idea of the algorithm is similar to straightforward LL(1) parsing with several major differences. The incremental algorithm must decide how to handle the situations when it advances to a satisfied token element but has a non-empty parsing queue and conversely when it empties the parsing queue but has unsatisfied productions in the parse tree. The second situation is handled in follow-the-cursor parsing by essentially doing nothing. We do not want to remove any further tokens from the parse tree so the algorithm simply leaves unsatisfied productions in the tree and displays them to the user as soft templates. In the first situation the algorithm needs to open up space in the parse tree to accommodate the elements of the parsing queue. This is done by invoking a conflict resolution algorithm described below. Following the description of the conflict resolution algorithm we will present two algorithms that together accomplish the incremental parsing desired. The first is the inner parsing algorithm that does most of the work and the second is the outer parsing algorithm that provides high level control.

CONFLICT RESOLUTION

In our parsing algorithm we will need to resolve a conflict if the element at the front of the parse queue cannot be parsed at the current parse position. The conflict can exist because there is already a token at Parse Position as described above or it can exist simply because the Queue Element does not fit into the terminal or non-terminal symbol at the Parse Position. The general algorithm would have grafted such an element as an error. That is not satisfactory here for two reasons. The first is that there are now non-terminal rooted subtrees on the Parse Queue as well as tokens. A subtree may not be parsable at this point but the tokens along its frontier may be. The second reason is that the algorithm does not have the guarantee that the subtree rooted at Parse Position is properly prepared to be parsed because it may not have deleted the entire subtree rooted at Royal Node in the Sweep algorithm.

The goal is to parse the elements of the parse queue by disrupting as small a region of the parse tree as possible. There is a conflict here because we want to parse the tokens in the parsing queue but we would like to keep the tokens that are on the tree intact if possible. Our solution to this is to give priority to the parsing of tokens before the cursor. This may mean dislocating tokens on the parse tree. If tokens are displaced, they are grafted to the tree as error nodes rather than moving them to the parsing queue.

We first present some definitions.

- As a generalization of the previous definition we define **Royal Node** is defined to be the highest node in the tree that has Parse Position as the first leaf of its frontier. If no such node exists then Royal Node is defined to be the node at Parse Position.
- **Decision Node** is defined to be the *lowest* node on the path from Parse Position to Royal Node that has the element at the front of the Parse Queue in its first set. If no such node exists then Decision Node is defined to be NULL.
- **List Node** is defined to be a node on the path from the Decision Node to the Royal Node (inclusive)

that is a list structured production. If no such node exists then List Node is defined to be NULL.

- **Nullable Node** is defined to be a node along the path from the Parse Position to the Royal Node that is nullable and has the element at the front of the Parse Queue in its follow set. If no such node exists then Nullable Node is defined to be NULL.

The Royal Node is the highest point in the parse tree where the token at Parse Position (or the token that previously was the first token of Parse Position) caused a decision to be made. The Decision Node, if it exists, is the lowest production along the path from Parse Position to Royal Node that the front of the Parse Queue can belong to. If the Decision Node exists then we can try to find a List Node. List Node is a place in the parse tree where a list production can be found. This makes it a place where we can wedge in a new production without tearing down any existing parse tree. At most one list node can be found because if there were two or more then there would be an ambiguous parse. Finally, Nullable Node is a node that can be nulled while still allowing the element at the front of the Parse Queue to be correctly parsed.

The algorithm for resolving the conflict is presented in figure 3. It first finds the four nodes described above. If List Node exists then the list production is expanded by an additional element using the **GraftNewList** subroutine. In the **StealProduction** subroutine the tokens in the subtree rooted at the node of the first parameter are grafted to the right as error nodes. The (tokenless) subtree rooted at the node is then deleted leaving an open non-terminal that is either nullable or has the element at the front of the parse queue in its first set. The final chance to avoid grafting an error token is if there is a non-terminal subtree at the front of the parse queue. In this case the nonterminal is removed and replaced with its children in the **Reduce** subroutine. This process continues until the algorithm has freed up a non-terminal in the parse tree or has emptied the parse queue.

INNER PARSING ALGORITHM

Figure 4 shows the inner parsing algorithm. This algorithm iterates through its parsing decisions until it runs out of tokens and/or runs out of open parse tree.

If the front of the parse queue and the predicted parse tree element at the current parsing position agree then the queue element is simply grafted onto the tree at the current position. The parse queue is then popped and the parse position advanced. It may be that there is not an exact match but that the queue element is in the select set of Parse Position. In that case the production

OuterParse

```
while (NOT Empty(ParseQueue)) do
    InnerParse(ParsePosition, ParseQueue);

    if ((Satisfied(ParsePosition))
    AND (NOT Empty(ParseQueue))) then
        ResolveConflict(ParsePosition);
    endif
endwhile

ErrorRecovery();
```

Figure 5: Outer Parse for Follow-the-Cursor Parsing

indicated is instantiated (there can be only one by LL(1) restrictions) and the Parse Position is advanced to the first element of the new production.

If neither of the above cases hold then the element at the front of the parse queue does not fit at the current position. The algorithm checks to see if there is a non-terminal subtree at the front of the parse queue that can be reduced. If this is not the case then it checks to see if Parse Position is nullable with Queue Element as a correct follow. If this is the case then the non-terminal at Parse Position is nulled and Parse Position advances. If none of the above cases holds then the conflict resolution algorithm is invoked.

OUTER PARSING ALGORITHM

The outer parsing algorithm provides high level control over the inner parsing algorithm. It resolves conflicts when Parse Position is advanced to a token and Parse Queue is not empty or Parse Queue is empty but Parse Position is a non-satisfied production element. The former case is handled by the conflict resolution algorithm. The latter case is allowed as a legal state in follow-the-cursor parsing because tokens to the right of the cursor are not taken to satisfy the parse position.

At the end of the normal parsing loop an error recovery algorithm is called. The Error Recovery algorithm is the only algorithm that is allowed to parse past the cursor. In follow-the-cursor parsing it is sometimes necessary to invoke the *Steal Production* process that grafts tokens as errors to the right of the current parse position. It is also possible that a token has been inserted which will resolve an error in the syntax of the user buffer if they were included in the parse. The idea of the Error Recovery algorithm is to probe into the error tokens directly past the cursor to see if these tokens can be parsed correctly.

An outline of the error recovery algorithm is presented

ResolveConflict(ParsePosition)

```
while ((NOT Empty(ParseQueue)) AND IsToken(ParsePosition)) do
    RoyalNode = FindRoyal(ParsePosition);
    DecisionNode = FindDecision(ParsePosition, RoyalNode, QueueElement)
    ListNode = FindList(DecisionNode, RoyalNode);
    NullableNode = FindNullable(ParsePosition, RoyalNode, QueueElement);

    if (ListNode != NULL) then
        ParsePosition = GraftNewList(ListNode, ParsePosition);
    elseif (DecisionNode != NULL) then
        ParsePosition = StealProduction(DecisionNode, ParsePosition);
    elseif (NullableNode != NULL) then
        ParsePosition = StealProduction(NullableNode, ParsePosition);
    elseif (IsNonterm(QueueElement)) then
        Reduce(ParseQueue);
    else
        GraftError(ParsePosition);
    endif
endwhile
```

Figure 3: Conflict Resolution Algorithm

InnerParse(ParsePosition, ParseQueue)

```
while ((NOT Empty(ParseQueue)) AND (NOT Satisfied(ParsePosition))) do
    QueueElement = Front(ParseQueue);
    if (QueueElement matches ParsePosition) then
        Graft(QueueElement, ParsePosition);
        Pop(ParseQueue);
        Advance(ParsePosition);
    elseif (Select[ParsePosition, QueueElement] != ERROR) then
        Instantiate(ParsePosition, Select[ParsePosition, QueueElement]);
        Advance(ParsePosition);
    elseif (QueueElement not a terminal) then
        Reduce(ParseQueue);
    elseif (Nullable(ParsePosition) AND (Follows(ParsePosition, QueueElement)) then
        NullProduction(ParsePosition);
        Advance(ParsePosition);
    else
        ResolveConflict(ParsePosition, ParseQueue);
    endif
endwhile
```

Figure 4: Inner Parsing Algorithm

ErrorRecovery

```
< Set Consistent Parse >
while (we have an error token) do
    if (token is parsable) then
        < Parse Token >
        if (Completed Structure)
            < Update Consistent Parse >
        endif
    else
        break;
    endif
endwhile

<Back up to last Consistent Parse>
```

Figure 6: Error Recovery

in 6. The algorithm begins by saving the current parse tree status, called the initial *consistent parse*. Each error token is then considered in turn. If the error token can be parsed correctly then that is done. If parsing the token completes a production in the parse tree then the consistent parse is updated to be the current parse state. The loop terminates when it runs out of error tokens or it encounters an error token that cannot be parsed correctly. It then backs up the state of the parse tree to the last consistent parse and exits.

INCOMPLETE GRAMMARS

Incomplete grammars presented here introduce two new non-terminal classes, *unstructured*[2] and *preferred* non-terminals, into language description grammars. Preferred non-terminals are the left-hand-sides of a special production class called preferred productions. Intuitively, the unstructured non-terminal class allows the language designer to have a production that escapes the structuralization process. A preferred production is a way of finding structure within the lack of structure of the unstructured non-terminal.

A conventional LL(1) grammar can be described as a tuple [PLRS76]

$$G = (S, T, N, P)$$

where

S is the start symbol of G, $S \in N$.

T is a finite set of terminal symbols.

N is a finite set of non-terminal symbols.

P is a set of production rules.

An incomplete LL(1) grammar is described as a tuple

$$G = (S, T, N, U, P, P_U)$$

where S, T, N, and P have their conventional meaning and

U is a distinguished set of non-terminal symbols denoted *unstructured*, $U \in N$.

P_U is a distinguished set of production rules *denoted preferred* productions, $P_U \in P$.

An unstructured non-terminal can occur at any point in the right-hand-side of a production rule. For the purpose of constructing the *select* sets of normal non-terminals (non-terminals that are not unstructured non-terminals) each occurrence of an unstructured non-terminal is treated as a unique, distinguished terminal symbol T_i, $T_i \notin T$. Thus a non-terminal's select set will contain an entry for each terminal symbol in its first set and an entry for any unstructured element that it can be derived from it. This is similar to the way that non-terminals are treated in incremental parsing. For parsing purposes we do not construct the *first* set of an unstructured element but we do construct the *follow* set of an unstructured element in the normal way. We do not construct the first sets for unstructured elements because their first sets vary at parse-time, depending on the shape of the parse tree. Intuitively, the run-time first sets vary because we want the unstructured element to act as a wild card non-terminal and accept any token that is not otherwise accepted at the point that the unstructured element occurs.

Consider, for example, the grammar:

```
A : a
  | C
  ;

B : b
  | C
  ;

C : Unstructured
  ;
```

If we are currently focussed at non-terminal A, we want any token except "a" to lead into production C. If we are focussed at non-terminal B, then we want any token but "b" to be accepted by C. Thus, the meaning of the same unstructured element (and by side-effect,

[2] Orailoglu refers to this non-terminal class as *Unknowns*.

C) will changed at run-time depending on the current parsing context when it is encountered.

A preferred production is a production that can find structure within an unstructured non-terminal. Its first set is calculated as for normal productions rules. Because the preferred production can be followed by the resumption of the unstructured non-terminal then the follow set should be anything that does not cause conflict with the preferred production. Thus if p $\in P_U$, y = left-hand-side(p),

$$\text{Follow(y)} \equiv \text{Can-Legally-Follow(y)}$$

where Can-Legally-Follow is a relation that generates the set of all tokens that can follow a non-terminal without causing a parsing conflict with that non-terminal.

TRANSFORMATIONS

Orailoglu devised specialized algorithms to parse based on incomplete grammars. This section will show how to transform an incomplete grammar into a complete grammar that can be parsed with conventional LL(1) algorithms. The obstacle to the traditional parsing of incomplete grammars has been that the first set of an unstructured element effectively changes at run-time depending on the state of the parse tree where the unstructured element is introduced. It will be shown that the decisions in Orailoglu's implementation which are made at run-time, can be predicted at the time the incomplete grammar is analyzed. This allows the incomplete grammar to be transformed into a complete grammar that recognizes the same language.

A simple example is presented to show the flavor of the material that will follow. Consider the incomplete grammar:

```
A : a c
  | b c
  | U (an unstructured element)
  ;
```

The token set of the grammar is {a, b, c, ERROR}. The intent of the grammar writer is clearly that a leading token of *a* will invoke the first right-hand-side, a leading token of *b* will invoke the second right-hand-side, and any other token will invoke the third right-hand-side because of the unstructured element. Thus the first set of the unstructured element is effectively {c, ERROR} and as a result the first set of non-terminal A is the entire token set.

Now consider the grammar:

```
A : b B          B : b
  | c C            | D
  ;                ;

C : c            D : d
  | D              | U
  ;                ;
```

The token set of this grammar is {b, c, d, ERROR}. The intent of the unstructured element in the grammar varies with the shape of the parse tree. If the current non-terminal is B then any token in the set {c, ERROR} will derive the unstructured element in D but if the non-terminal is C then any token in the set {b, ERROR} will derive the unstructured element. The thing to note is that this can be predicted at the time that the grammar is analyzed.

The above grammar can be transformed into the grammar:

```
A : b B          B : b
  | c C            | D
  ;                ;

C : c            D : d
  | D              | U1
  ;                ;

U1 : Ua ( Ub )*
   ;

Ua : b           Ub : b
   | c                | c
   | ERROR            | d
   ;                  | ERROR
                      ;
```

This grammar has the same token set as the previous grammar. The only difference is that three new productions are introduced to represent the structure of the incomplete element. The first production gives the conceptual structure of the incomplete element. The second production represents tokens that can occur first in the unstructured element and the third production represents what may follow the first element as the body of the unstructured element. Notice that *U1* contains any token that is not otherwise in the first set of D. This causes the grammar to be ambiguous because the token b is in the first set of both alternatives of non-terminal B and the token *c* is in the first set of both alternatives to non-terminal C. The key to the transformation method is to resolve the conflict in each case in favor of the alternative that does *not* derive the unstructured element. With this method of resolving the parsing ambiguity, the transformed grammar recognizes exactly the same language as the untransformed grammar.

The above example illustrates the spirit of the transformation method on a very simple grammar. The remainder of this section will show that the method can be applied to any incomplete grammar of the form described by Orailoglu [Ora83]

For parse table calculations each unstructured non-terminal is recognized as a separate production but is treated somewhat differently when checking LL(1) grammar restrictions. Although they are technically different elements, unstructured elements must satisfy some restrictions as if they were the same terminal. Two distinct unstructured elements cannot both occur in first set of a production or in the follow set of a production. There are also restrictions to avoid ambiguity. An incomplete element cannot be followed by another incomplete element, and incomplete elements can neither start nor end preferred productions. If a token is both in the first set of a preferred production and the follow set of an unstructured element then the conflict will be resolved in favor of the follow set. No token may appear in the first set of more than one preferred production because this would cause a grammar ambiguity.

An unstructured element may be legally derived at run-time if all of the following conditions apply:

- The current parsing position is a non-terminal that can derive the unstructured element in the grammar
- The current parse queue element is a token that is not in the select set of the current non-terminal.
- The current non-terminal is not nullable with the input token in its follow set[3].

If all of the above conditions apply then the tree is expanded to derive the unstructured element and the algorithm enters unstructured parsing mode. While in unstructured mode the parser accepts any token as part of the incomplete element until it receives a member of the follow set of the incomplete element or a member of the first set of a preferred production. If a member of the follow set is encountered then the incomplete element is closed. If a member of the first set of a preferred production is encountered then the preferred production is instantiated and parsed normally, and unstructured parsing is resumed when it completes.

The transformation approach will be to replace each unstructured element U by a non-terminal $U\prime$ which is the left-hand-side of a production rule of the form

$$U\prime \;:\; U_s\ (\ U_b\ )^*$$
$$;$$

where U_s derives tokens and preferred non-terminals that may start the unstructured element and U_b derives the set of tokens and non-terminals that may be in the body of the unstructured element.

The production rule for U_b is the easier of the two to calculate. The first step is to calculate the follow set of U in the normal manner. This calculation is already performed by the existing algorithms. This tells what *not* to include in the token set derivable from U_b. Let the set of preferred non-terminals be denoted P = p_1, ... , p_n and let

$$F \;=\; T - follow(U) - first(p_1) - \ldots - first(p_n)$$

Then the production rule for U_b is

$$\begin{array}{lll} U_b & : & t_1 \\ & | & \ldots \\ & | & t_k \\ & | & p_1 \\ & | & \ldots \\ & | & p_n \\ & ; & \end{array}$$

where t_1, ..., t_k are the elements of F.

This production correctly parses the internal part of the incomplete element because it derives all the preferred productions and all tokens not in the first set of a preferred production or in its follow set. If there is a conflict between the first set of some p_i and the follow set of U then, as before, the conflict is resolved in favor of the follow set.

The calculation of how unstructured elements can be derived involves not only calculation of the production rule for U_s but also the rules for resolving conflicts that arise in the select tables of the grammar. An unstructured element occurs in the right-hand-side of a production of the form

$$\begin{array}{lll} A & : & w\ U\ x \\ & | & rhs_2 \\ & | & \ldots \\ & | & rhs_n \\ & ; & \end{array}$$

where w and x may each be empty and where $n \neq 1$. Thus the simplest production rule containing an unstructured element is of the form

$$\begin{array}{lll} A & : & U \\ & ; & \end{array}$$

The first step in calculating a production rule for U_s is determining whether w is nullable. Let F be the set of tokens that can occur in the first set of U. If w is not

[3]This slight variation from Orailoglu's implementation is introduced to give a more consistent treatment of unstructured elements.

nullable then set F to the entire token set. Any parsing conflicts with w will be resolved in the parse table construction phase. If w is nullable then F must be calculated so that it does not cause a parsing conflict with w or with any other right-hand-side of the production rule. Thus, the lead-in to U can be

$$F = T - first(w) - first(rhs_2) - \ldots - first(rhs_n).$$

The set F is the select set of U for parsing purposes. This will keep members of the first set of a preferred production that are not in F from interfering with calculation of the select table. The set of tokens that can lead directly to U is then

$$F - first(p_1) - \ldots - first(p_n) = t_1, \ldots, t_j$$

and the production rule U_s is

$$\begin{array}{lll} U_s & : & t_1 \\ & | & \ldots \\ & | & t_j \\ & | & p_1 \\ & | & \ldots \\ & | & p_n \\ & ; & \end{array}$$

where some of the p_i may not be derivable because no member of their first set is a member of F. The is allowable because the first set of U has already been calculated.

Using F as the first set for U_s guarantees that the production $U\prime$ will not cause a parsing conflict with the first sets of the right-hand-sides of the production in which it occurs, but it may still cause a conflict in productions that can derive A. The key to the transformation method is to always resolve the ambiguity against the alternative that derives the unstructured element. The first step of this is to calculate the select table and follow sets in the usual manner, using the designated first sets for the transformed elements. Next comes the grammar validity check.

If there is a first-first conflict in the grammar then check to see if one of the alternatives derives a transformed unstructured element. If so, resolve the conflict by selecting the other alternative. If there is a first-follow conflict caused by the first set of an unstructured element in the follow set, remove the conflicting token from the first set of the following non-terminal that derives the unstructured element. If there is a first-follow conflict caused by an unstructured element in the first set of a non-terminal, then remove the token from the first set of the non-terminal that derives the unstructured element. The first-first conflicts should be resolved before the first-follow conflicts so that the problem of multiple conflicts does not arise. Note that all of these conflicts do not occur in the parse table construction for a parser that treats incomplete grammars specially because the unstructured elements are treated essentially as distinguished unique tokens in the grammar analysis.

The purpose of the above conflict resolution strategy is to make the decisions when the parse tables are built that the parser would make at run-time in a parser for incomplete grammars. To see that this is true, first consider the production U_s in the case where w in the grammar above is non-nullable. In an unstructured parser the incomplete element will be encountered and instantiated when w completes, i.e., when the parser encounters a legal follow of w. This is exactly what happens in the transformed grammar.

Suppose that w is nullable. Then the unstructured element can be derived directly by A and indirectly by productions that derive A. Assume that the current non-terminal is A. The unstructured element will be directly derived if the current token is not in the first set of w or the first set of any other right-hand-side of A, and if A is not nullable with the current token in the follow set. The same action is taken in the transformed grammar because $U\prime$ does not have any members of the first set of w or the other right-hand-sides in its first set.

Now assume that the current non-terminal is not A but one that can derive A. In the unstructured parser, the unstructured element in A can be derived if the current token is not in the first set of the current non-terminal and if the current non-terminal is not nullable with the token in its follow sets. These are exactly the conditions under which $U\prime$ can be derived in the transformed grammar. Tokens that would not derive the unstructured element above will not do so in the transformed grammar because of the manner in which parsing conflicts are resolved in the select table. The tokens that are left are those that do not cause conflicts and they derive the unstructured element.

The last point to establish is the validity of the grammar model in which the incomplete element was introduced. The model is valid because only one unstructured element needs to be concentrated on at a time. This is true because

- A non-terminal cannot have two separate unstructured elements in its first set.
- An unstructured element cannot have an unstructured element in its follow set.
- A preferred production cannot start or end with an unstructured element.

It has been shown that an incomplete grammar may be transformed into an equivalent complete grammar. Is

there any advantage in doing so? The grammar transformation introduces new productions and thus causes the parsing tables to increase in size. This will in turn cause the run-time parse tree data structured to grow in size. The transformed grammar will introduce approximately one extra parse tree node for each token that is parsed as part of an unstructured element. The transformation process also significantly increases the complexity of the grammar analysis process. The real advantage of the algorithm is that it allows the incomplete grammar to be parsed by a conventional LL(1) parser. This is an advantage because it makes the grammars more easily adapted to other parsers and because it reduces the complexity of the parsing algorithm.

PREVIOUS WORK

Syntax-directed editors such as the Cornell Synthesizer [RT84, TR81] allow phrases to be entered as text below some level in the syntax. Textual input is parsed by a stand alone bottom-up parser that begins with the nonterminal represented by the current placeholder. The parsed text must be able to be grafted onto the parse tree as a complete, correct subtree.

Carlo Ghezzi and Dino Mandrioli have developed a bottom up parsing algorithm with is based on the use of grammars that are both LR and RL [GM79b, GM79a]. The authors also have published an algorithm that is more complex but operates on a more general class of LR grammars [GM80]. The BABEL editor [Hor81] is based on the Ghezzi and Mandrioli symmetric algorithm. Programs are not permitted to be incomplete, and it is not possible to place unexpanded placeholders in the tree. Kirslis [CK84, Kir85] has extended the Ghezzi and Mandrioli LR(0) algorithm to LR(1), has modified the parsing algorithm to handle comments and introduced explicit error handling routines.

An editor dubbed SRE for Syntax Recognizing Editor has been developed at the University of Toronto [BHZ85]. This editor provides flexible error handling by dividing the parser function into two levels. A low-level parser guarantees that the user's program consists of a sequence of syntactically correct lines. A high-level parser guarantees that the syntactically legal lines form a syntactically legal program. Only low-level syntactic correctness is enforced while text is being entered. Syntax errors within lines are pointed our immediately and the user is forced to correct them before proceeding. Syntax errors between lines are only pointed out when the user requests a high-level parse. Morris and Schwartz [MS81] published a LL(1) parsing algorithm that maintains a sequence of syntactically correct parse trees.

Orailoglu implemented an LL(1) incremental parsing algorithm as part of the the restructuring programmable display editor (RPDE, now called Fred) at the University of Illinois [Ora83, Shi85]. The algorithm maintains a single parse tree but allows multiple errors with unrestricted parsing by invoking a simple context (and history) sensitive error recovery algorithm. The key disadvantage of the algorithm is that it lacks an effective means of limiting parsing and tends to parse forward too far, recovering from errors along the way, when changes are made to the internal structure of a program. Orailoglu [Ora83] provided the original implementation of incomplete grammars.

CONCLUSION

This paper presents an incremental LL(1) parsing algorithm that is suitable for use in language-based editors and that has been implemented in *Fred*, structured, screen-based editor. A keystroke intensive mode of user interaction motivates the follow-the-cursor style of parsing in which parsing is normally halted at the cursor, leaving suspensions in the parse tree that are indicated to the user as soft-templates. Algorithms for tree preparation, incremental parsing, and error recovery are presented. The algorithms implement a style of user interaction that is both efficient and convenient. It is efficient because the editor only needs to perform limited parsing after changes. It is convenient because the user is able to enjoy the benefit of structuralization while retaining complete freedom of program entry.

Incomplete LL(1) grammars are presented as a way of dealing with the complexity of full language grammars and as a mechanism for providing structured editor support for task languages that are only partially structured. Orailoglu devised specialized algorithms for parsing based on incomplete grammars. This work shows how the grammars can be translated into conventional LL(1) grammars, eliminating the need for specialized parsing algorithms.

References

[BHZ85] Frank J. Budinski, Richard C. Holt, and Safwat B. Zaky. Sre - a syntax recognizing editor. *Software-Practice and Experience*, 15(5):489–497, May 1985.

[CK84] Roy H. Campbell and Peter A Kirslis. The saga project: A system for software development. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984. (Released as ACM SOFTWARE

ENGINEERING Notes 9(3) and ACM SIGPLAN Notices 19(5).).

[GM79a] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–579, July 1979.

[GM79b] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, July 1979.

[GM80] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3), July 1980.

[Hor81] M.R. Horton. *Design of a Multi-Language Editor with Static Error Detection Capabilities.* PhD thesis, University of California, Berkeley, July 1981. ERL Technical Reprot 81/53.

[Kir85] Peter A. Kirslis. *The SAGA Editor: A Language-Oriented Editor Based on Incremental LR(1) Parser.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1985.

[MS81] J. Morris and M. Schwartz. The design of a language-directed editor for block structured languages. *SIGPLAN Notices*, 16(6):28–33, June 1981. Proceedings of ACM SIGPLAN/SIGOA Symposium on Text Manipulation, Portland.

[Ora83] A. Orailoglu. *Software Design Issues in the Implementation of Structured Editors.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1983.

[PLRS76] II P.M. Lewis, J. Rosenkrantz, and R.E. Stearns. *Compiler Design Theory.* Addison-Wesley, 1976.

[RT84] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984. (Released as ACM SOFTWARE ENGINEERING Notes 9(3) and ACM SIGPLAN Notices 19(5).).

[Shi83] John J. Shilling. Improvements to a structured, screen oriented editor. Technical Report Report No. UIUCDCS-R-83-1155, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1983.

[Shi85] John J. Shilling. Fred: A program development tool. In *Proceedings of SOFTFAIR II (San Francisco, California, December 3-5, 1985)*, December 1985.

[Shi86] John J. Shilling. *Automated Reference Librarians for Program Libraries and Their Interaction with Language Based Editors.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1986.

[TR81] T. Teitelbaum and T. Reps. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9), September 1981.

Linguistic Information in the Databases as a Basis for Linguistic Parsing Algorithms.

Apollonskaya Tatiana A.

Beliaeva Larissa N.

Piotrowski Raimund G.

Applied Linguistics Department

Herzen Pedagogical Institute

Moika emb. 48

191186 Leningrad USSR

The focus of this paper is investigation of linguistic data base design in conjugation with parsing algorithms. The structure of linguistic data base in natural language processing systems, the structure of lexicon items and the structure and the volume of linguistic information in automatic dictionary is the base for linguistic parsing organization.

The avalanche-like flow of documents in natural Languages (NL) calls for a reliable cybernetic means to conduct its intellectual processing and formalized catalogization and classification.The most effective instrument helping to achieve these tasks is Linguistic Automaton (LA). LA is an all-round complex of hard-, soft-, lingua-, and partly tutorware.

During recent years, the linguistic research activity at Leningrad Speech Statistics Group (SpStGr) on natural language processing was concentrated on the pursuit of two objectives:

first, the lexico-semantical,morphological and pragmatical problems of automatical dictionary (AD)

and second, the construction of parsing programs.

At the same time, it had long been asserted that semantic and pragmatic information contained in AD and in LDB must be used to resolve many of the lexical and grammatical ambiguities that occur in the text. The adequate resolution of ambiguities is often critical to the MT process,since often ambiguities which occur in source language cannot be maintained in target language.

The creation of such a complex needs, on one hand, extensive theoretical investigations in the field of systemic linguistics and consideration of possible practical contributions in such diverse natural language processing (NLP) areas as machine translation information retrieval,indexing,automatic abstracting etc.On the other hand, all these systems need special parsing algorithms and special structure of automatic dictionary (AD).

The conjugation of AD structure and parsing hierarchy is the focus of this paper. This conjugation is hindered by a series of antinomies, the principal of which are two paradoxes:

1.The linearization paradox consists of non-additivity of text undestanding while human text processing.The process of text undestanding is sumultaneous with text reception. When modelling this process on computer, mental sumultaneous-associative processes are successively linearized during parsing.

2. The static and dynamic paradox consists of the necessity to model the dynamically and constantly enriching process of text generation and reception during the human intellectual activity with the help of previously fixed procedures on the basis of a static model of averaged professional competence, stated in LDB.

As a matter of fact, the creation of NLP system is a process of gradual overcoming these paradoxes. The success of such a process is determined by:

- the correctness of the elaborated models of professional competence;
- the database organization model and the professional competence model level;
- the level of the model of language competence, and correspondingly,
- the level of linguistic algorithms and program elaboration;
- the optimum of parsing realization;
- the level of computer development.

Thus, when designing NLP system it is necessary to conjugate the three previously established models in a united technological structure which allows to minimize the influence of the described paradoxes on the NLP rezult.

The basis of this conjugation is both the organisation of data processing (parsing) and the organisation in LDB.

The LDB organization must answer to the next requirements:

1) the data, which are inserted in the LDB, and the data descriptions must be structurized in accordance with the procedures, which are realized in a specific NLP system;
2) the LDB must be organized optimally concerning the problems, which the specific NLP system is tuned on.

The optimum LDB organization requires a modular design which consists in realization of LDB as a set of nonrigidly-linked modules. This modularity allows to arrange a LDB as modules are ready and eliminates the data duplication. Besides, it allows the step-by-step solving of NLP problems.

Besides, we must orient the structure of the system as it is and the structure of the linguware on system pragmatics which demands to investigate

- the specialist needs (express-information, signal translation, high-quality post-edited translation);
- the details of information flow (document types, volume of document and document flow, source language types, possibility of pre-, inter- and post-editing)
- the peculiarities of terminology and syntax of a special domain.

The organisation of LNP system implies the systemic principle, that determines conditions of

- the description of lexicon and morphology of source and target languages,
- the description of source and target languages syntaxis;
- the interface between LDB and software.

In accordance with this we can establish the main principles of MT system design. They are as follows:

1. The principle of modular and hierarchical organisation.
2. The principle of separation of basic and problem-oriented modules of lingua- and software.
3. The principle of the transfer as the translation process basis.

The main feature of our LA design approach is a tend to separate the groups of interconnected processes in a complicated ATP process as a whole. This separation is to be done so that their interaction both give certain system stability for different input data and allow to preserve open modular structure.

At the same time these principal points in NLP system development inevitably lead to dimention crisis. That's why in the eleborated system the hierarchy of translation levels is clearly defined. The development of the hierarchy structure of the system is realized in a descending line, "from top to bottom". This point of view implies the following:

- the exact analysis levels definition and the levels hierarchy ascertainment;
- the volume and goals definition, that means the definition of the goal of each analysis level from above, the definition of information volume of a word entry and of information distribution in word areas;
- the availability of an open modular system.

In accordance with this the procedure of translation is devided into subprocesses (levels) each having its own functional value. The results of development of each level form the basis for processing on a higher level. Thus a phrase level, a sentence level, a functional component level, a functional unit level, a lexical unit level are separated. Each level is connected with the translation process. Translation is regarded here as a multi-level process, each of its procedures translates a component of the special level.

It means that the source structures of each level are transformed into output structures which may be modified on a higher level in accordance whith the structural features of this higher level.

Thus the translation process is simulated in the system in question as a composition of lexical and semantic-syntactic translation process.During the lexical translation process the identification of text and dictionary units and the extraction of dictionary information from the lexicon blocks are carried out.During semantic-syntactic process the interlanguage structure transfer which uses the whole information received on the lexical translation phase and joins up grammar and semantic LDB blocks is carried out.This transfer process is simulated as an aggregate of verticaly conjugated subsystems, the hierarchy of the components which are extracted from the text.

The Linguist's aim in this conception of translation process is to define all the levels of translation and analysis, to formulate the set of characteristics which are necessary for the source structure modification into the target structure of the definite level and to definite the specification of the next higher levels.

Proceeding from the stated idea of the NLP system design let's analyse the structure of AD and the reciprocal correlation of grammar and dictionary on each of the determined levels in the analysis and translation of the predicate of the sentence.

During the verb entry elaboration it is necessary to choose the most important, key structural elements (which determines the dictionary volume), and to state a set of rules for the singled out linguistic elements functioning (which determines the grammar volume and the principles of parsing).

For a multilanguage ATP system the choice of AD item is determined both by word- and formbuilding principles different in specific languages as well as by the representation features of semantic text items. Besides that the choice of a basic dictionary item is determined by the tasks of NLP system and the LDB universality level.

In the Soviet NLP systems the Russian language is used as a metalanguage for source text definition as well as the target language. The unity of the target language enables to unify its definition for all NLP systems from foreign languages into Russian and to unify the procedures of morphological synthesis of a Russian wordforms.

When we design MT system for translation from the Russian the procedures of the morphological analysis are unified as well. In any case machine morphology definition of the Russian language constitutes a separate module and is used in all versions of the system.

SILOD-MULTIS AD includes source word dictionaries, which are organized as dictionaries of word usages and dictionaries of stems, source phrase dictionaries,target stems definitions and machine morphology for different languages.

Any AD that characterizes a specific language includes a universal structure set of dictionary items and machine morphology. All the source language ADs have the same function and a united scheme organisation.

This scheme allows to unify such procedures of the source language text processing as a selection of minimum text units, the morphological analysis, the identification of the text with AD items, the organization of the dictionary information file.
Any lexical unit (LU) in AD acquires a description on the morphological, syntactic, semantic and functional levels as an appropriate characteristic set.

The basic version of the system includes dictionary items (DI), which consist of the following characteristics:

-the head LU as it is: a stem, a word or a phrase;

-the lexical and syntactic code (LSC), which depends on the typological features of the source language, its grammar and parsing algorithms which are realized in the system in question;
-the translation, which is stored as references to the corresponding target language items (stems and lexical and grammatical characteristics).

For analytical languages the most expedient is the introduction of separate word forms, as it allows to increase the speed of the system while the growth of the dictionary volume is negligible. For synthetical languages machine stems are the head LU in the DI and the input AD is filled up with machine morphology.

In order to reduce the memory volume for AD location we resort to the artificial morphology transformation, i.e. to the insertion of the agglutinative morphology. The essence of the latter consists in the process of the selection in any word usage a machine stem and an affix "sticking"to it.

The concept of the inserting of machine affix allows to elaborate the Russian machine grammar, formed as a set of paradigms - machine affix chains. Each typical paradigm correlates with the grammatical characteristics of stems and the word formation mode. The link between a machine stem and a paradigm is realized with the help of a special code, which characterizes all the word forms which can be generated from the stem in question.

The use of this machine morphology allows to realize the wordform generation proceduress in accordance with the lexical and grammatical characteristics which are formed in the course of MT, and to make this procedure a universal one for any language pair.

Accordingly, the elaborated Russian stem dictionary permits to identify automatically the text words with dictionary items and to ascribe their morphological characteristics accurately to case homonyms. The result of morphological analysis, which is received with the help of LDB and special lexical and morphological analysis algorithms, is a source for parsing and transferring algorithms for Russian-English MT.

A two-layer system of lexical and semantic coding is realized in the LDB of SILOD-MULTIS system. The upper level of this coding is constituted by 30-element LSC which is formed in DI immediately. LSC formation is created in accordance with the coding tables elaborated for every source system languages. This information can be formed on-line.

The levels discussed above specify the lexical and grammatical description of LU in LDB. The syntactic definition covers the functional LU characteristics which determine their potential capacities to accomplish a specific role in syntactical sentence structure. The semantic definition which constitutes in a distinct, internal level is concerned with the transfer from the linguistic phenomena proper to the extralinguistic ones. The formation of this definition is based on the structural investigation of the domain, that is to be manifested.

Let's consider the structure of information on the example of verb entry of French-Russian MT system, which is the base for parsing system.

On the lexical level of the analysis the predicate equal to the morphological verb-form is development. In the French-Russian MT system the verb is presented in two ways: as word-forms for the irregular and suppletive verbs (avoir, etre, aller, vouloir) and as machine-stems with their standard paradigm.

Each source standard paradigm includes information sufficient to establish a link with a definite stem and a corresponding word entry (item).The analysis procedure is performed according to the

morphological tree.

On the functional unit level a verb and nominal segments are identified. The structures of this level include verb segments equal to the complex verb, tense of the pronominal verbs and of the verbs in active and passive forms. The procedure is performed on the information contained in various positions of theverb entry:

- the information of the verbs belonging to the auxiliary class are contained in the LSC. This information is necessary for the discrimination of the complex verb tenses. Position Six of the LSC of the verbs "aller", "venir" contains the information necessary for "Immediate" tenses identification;
- the passive form identification footholds on Position Eight (transitivity notes), but for its translation the corresponding rules of Position Eleven are to be used. This position contains the information of the possibility of the shortened passive participle form usage ("est ouvert" - opened), the pronounal form usage ("est prepare" -- is prepaired), the active form usage ("est suivi" - follows).

The pronounal form is translated according to the information of Position Twelve of the verb entry. The compound nominal predicate identification and translation is performed on the basis of Position Fourteen.

As to the designing of the grammar rules which direct the analysis and translation of impersonal construction it is prescribed by the information of Position Fifteen.

The inner verb class relations are of fixed character. This makes it possible to présent a verb segment as a frame including all verb-connected elements (the objective pronouns, the negative and limiting partycles) and verb elements (the auxiliary verbs and the participles of a conjugated verb) . During the analysis on the functional segment level the procedure of homonymy elimination is realized.

The result of the procedure on this level is a chain of source and target functional segment. Together with this the target functional segment (a verb group) gets a certain set of indications necessary for the next level - the sentence level analysis.

The peculiarity of verb elements analysis is their immediate functioning on the sentence level, as to the nominal groups, they have an additional stage - the stage of functioning components formation. This is explained by the diversity in the interrelations of the nominal group elements.

Thus up to the begining of the sentence level analysis the structure of the verb functional segment is known, the ways of the given verb structure presentation are defined; the verb elements homonymy is eliminated. The designed output structure gets the total set of indications necessary for its analysis on the sentence level.

This set is compiled of the active form verb entry information:

- the indication of the obligatory direct object according to Position Eight;
- the indication of the possible information distribution according to Position Six;
- the indication of the possible object or adverbial modifier according to Position Nine.

This set is also compiled of the information ascribed to the pronominal verbs (the type of government), according to Position Thirteen, and to the passive form verbs according to Position Ten; and the indications formed in the translation process on the preceding levels of the analysis (tense, number, person and others) of the compound verb constructions in all mentioned forms.

By the sentence level analysis stage a number of "refusals",

got on the previous levels, are piled because of various causes (ambiguity of the structure in a bilingual situation, uneliminated homonymy, impossibility of the analysis on the preceding stages of a number of constructions (infinitive, passive, impersonal, pronominal) requiring the subject-object transfoormations for a correct translation). Thus it is possible to pass over to the choice of the translational structure of the whole sentence only after the functional of the nominal and verb groups as sentence members is defined.

While choosing the translational equivalent on the sentence level some difficulties arise in the case of the input and output structures inadequacy.

Then it is possible to resort to the subject-object transformations. The subject-object transformations may be realized either with the help of the sentence members rearrangement or by the case forms of the target structure change or by the conversives search.

The conversives search practically leads to the increase of the number of the verb translational equivalents. More productive is the way of subject-object transformations, connected not with the sentence members rearrangement but with the case relations change in the output structure. The results of the sentence level elaboration is the obtaining of the output sentence structure.

On the phrase level the translation of the whole complex sentence is performed. Here the subordinate clause translation is corrected. In particular the testing of the correct choice of the conjunctions and relative pronouns, introducing the subordinate clauses. Thus for a correct choice of the translational equivalent of an homonymous form "que" (what, so that, which) it is necessary to resort to Position Eight of the word entry information. The information contained in it gives an opportunity to choose the corect form (indicative or subjunctive) for the subordinate clause verb translation. The same process takes place when translating the subordinate clause with "clout". The correct choice of the translational equivalentfor the whole subordinate clause is realized only with the orientation to te indication of Position Nine of the main clause verb.

Thus the chosen point of view on the MT system elaboration makes it possible to realize the whole volume of the research goals. In this circumstance that is an indipensable facility for the designing of the interaction of grammar and dictionary on each of the system levels.

Hence this conception creates the necessary faciolities for the development of the systems forecasting the analysis of newly arising situations on the basis of the once elaborated situations.

W.J.Hutchins. Machine Translation: Past,Present,Future. Chichester: Ellis Horwood , 1986;382p.

Ju.Kondratieva, R.Piotrowski, S.Sokolova. Organization of the Russian-English MT-algorithm. SCCAC Newsletter, No 4, Bowling Green, 1988, p.21.

N.Maruyama, M.Morohashi, S.Umeda, E.Sumita. A Japanese sentence analyzer. -In: IBM Journal of Research and Development, Vol.32, No 2, March 1988, pp.238-250.

S.Nirenburg(ed). Machine Traslation: Theoretical and Methodological Issues. Cambridge: Cambridge University Press, 1987. -350p.

SILOD. A Russian-English Translation Support. New-Delhi: Elorg-Computronics India, 1988. -16p.

February 13, 1991

Session C

BINDING PRONOMINALS WITH AN LFG PARSER

Rodolfo Delmonte°
Dario Bianchi *
° University of Venice - Department of Linguistics - Ca' Garzoni Moro - San Marco 3417 - 30124 VENICE(It)
*University of Parma - Department of Physics - Viale delle Scienze - 43100 PARMA(It)
°E-mail: delmonte@IVEUNCC.bitnet
°VAX:MG9VEVB3@icineca
°Phone: 39-41-5298459/5204477/5287683
°FAX: 39-41-5211559

Abstract

This paper describes an implemented algorithm for handling pronominal reference and anaphoric control within an LFG framework. At first there is a brief description of the grammar implemented in Prolog using XGs(extraposition grammars) introduced by Pereira(1981;1983). Then the algorithm mapping binding equations is discussed at length. In particular the algorithm makes use of f-command together with the obviation principle, rather than c-command which is shown to be insufficient to explain the facts of binding of both English and Italian. Previous work(Ingria,1989;Hobbs,1978) was based on English and the classes of pronominals to account for were two: personal and possessive pronouns and anaphors - reflexives and reciprocals. In Italian, and in other languages of the world, the classes are many more. We dealt with four: a.pronouns - personal and independent pronouns, epithets, possessive pronouns; b.clitic pronouns and Morphologically Unexpressed PRO/pros; c.long distance anaphors; short distance anaphors. Binding of anaphors and coreference of pronouns is extensively shown to depend on structural properties of f-structures, on thematic roles and grammatical functions associated with the antecedents or controller, on definiteness of NPs and mood of clausal f-structures. The algorithm uses feature matrixes to tell pronominal classes apart and scores to determine the ranking of candidates for antecedenthood, as well as for restricting the behaviour of proforms and anaphors.

1. The parser

A parser is presented which works on Italian and German, and binds pronominals within their utterance leaving unsolved the reference of free pronouns. It is divided into two main modules, the grammar and the binding algorithm. The grammar is equipped with a lexicon containing a list of fully specified inflected word forms where each entry is followed by its lemma and a list of morphological features, organized in the form of attribute-value pairs. Once the word has been recognized, lemmata are recovered by the parser in order to make available the lexical restrictions associated to each predicate. Predicates are provided for all lexical categories, noun, verb and adjective and their description is a lexical form in the sense of LFG. It is composed both of functional and semantic specifications for each argument of the predicate: semantic selection is operated by means both of thematic role and inherent features. Moreover, in order to select appropriately adjuncts at each level of constituency semantic classes are added to more traditional syntactic ones like transitive, inaccusative, reflexive and so on. Semantic classes are meant to capture aspectual restrictions which are crucial in deciding for the appropriateness and adequacy of adjuncts, so that inappropriate ones are attached at a higher level.

Grammatical functions are used to build f-structures and processing pronominals. They are crucial in defining lexical control: as in Bresnan (1982), all predicative or open functions are assigned lexically or structurally a controller. Lexical control is directly encoded in each predicate-argument structure.

Structural information is essential for the assignment of functions such as TOPIC and FOCUS. Questions and relatives, (Clitic) Left Dislocation and Topicalization are computed with the Left Extraposition formalism presented by Pereira(1981;1983). Procedurally speaking, the grammar is implemented using definite clauses. In particular, Extraposition Grammars allows for an adequate implementation of Long Distance Dependencies: restrictions on which path a certain fronted element may traverse in order to bind its empty variable are very easily described by allowing the prolog variable associated to the element in question - a wh- word or a relative pronoun - to be instantiated in a certain c-structure configuration. Structural information is then translated into functional schemata which are a mapping of annotated c-structures: syntactic constituenty is now erased and only functional attribute-value pairs appear; also lexical terminal categories are erased in favour of referential features for NP's determiners, as well as temporal and modal features. Some lexical elements disappear, as happens with complementizers which are done away with and substituted by the functional attribute SCOMP or COMP i.e., complement clause.

From a theoretical point of view, using Prolog and XGs as procedural formalism we stuck on to LFG very closely (see Shieber(1985); Pereira & Shieber(1984); Pereira(1985))even though we don't use functional equations: in particular the Fusion mechanism can be performed straightforwardly and the Uniqueness Condition respected thanks to Prolog's unification mechanism. It differs from LFG's algorithm basically for dismissing functional equations: however,

functional schemata can encode any kind of information in particular annotated f-structures, keeping a clear record of all structural relations intervening between constituents. In particular, long distance dependencies are treated using XGs, since they can easily encode paths from a controller to its controllee, as well as restrictions to prevent "island violations". In this case, we don't rewrite an empty category by means of a rewriting rule, as in LFG, rather, we activate a procedure as in Pereira(1983): moreover, the bindee or controllee to be bound by its controller or binder is assigned semantic and functional features by its predicate so that semantic compatibility can be checked when required, or else features transmitted to the controller once binding has taken place: Italian is a highly structurally ambiguous or undetermined language (see Delmonte, 1985), so that semantic or thematic checking seems necessary at this level.

2. Theoretical Background

Italian has three reflexive elements, one of which is a possessive anaphoric pronoun, "proprio", than a short distance reflexive pronoun, "se stesso", and a long distance one "sè". The short distance reflexive "se stesso" has a distribution that is somewhat similar to the English reflexive "himself", though there are differences between the two. It may corefere with a coargument and its antecedent must appear in the same minimal finite domain. On the contrary with the long distance reflexive "sè" the antecedent must be a subject: however it must be "governed" by a preposition, i.e. it must be contained in an OBLique or an ADJunct PP. As to the long distance possessive anaphoric pronoun "proprio", it is subject oriented and clause bound, but in lack of an adequate antecedent it may look out of its clause (complement or adjunct or coordinate) for its antecedent. In addition, there is the multivalued clitc "si" which may be assigned the following functions "passivizing", "reflexive", "impersonal or arbitrary": its behaviour is determined strictly by the verb predicate to which it is bound. None of the reflexive elements may be used as SUBJects.

Italian has also four pronominal elements, one of which a possessive pronoun, "suo", than a Null Subject pronoun which behaves very closely to the English personal pronouns; finally a set of lexical independent pronouns which are used for contrastive or emphatic aims. All these pronouns look for their antecedent outside their minimal containing clause. As to the possessive "suo", it behaves quite differently from the corresponding English "his". "His" can be bound by an OBJect coargument, when it is contained in the SUBJect NP as for instance in "His daughter loves John". This is not allowed in Italian, the SUBJect being a strong domain for reference. The same applies to "proprio", which being a possessive anaphoric pronoun is sensitive to the grammatical function it is contained in. However, there is one exception, and this is the case constituted by psychic verbs, whose SUBJect is characterized by a thematic role which is very low in the hierarchy of theta-roles: it is an (emotional) Theme, as for instance in "La propria salute preoccupa ognuno/Gianni". Coreference between "proprio" and "ognuno" is allowed, but is banned with "Gianni" as antecedent. Clearly this does not apply to the corresponding "La sua salute preoccupa ognuno/Gianni" where no such coreference is allowed.

As Dalrymple(1990) comments, "constraints on anaphoric binding are lexically associated with each anaphoric element. In fact generalizations have been noted that deal specifically with the lexical form of the anaphoric element: elements of a particular morphological form are usually or always associated with particular sets of anaphoric binding constraints"(ibid.,2). It is interesting to note that such functional notions like "subject", "tense" and "predicate" are essential in defining these constraints, they all "denote some syntactically or semantically 'complete' entity"(ibid.3). As Dalrymple comments, "In a complete, consistent f-structure, a PRED denotes a syntactically saturated argument structure; presence of a SUBJ entails a predication involving some property and the subject; and presence of TENSE indicates an event that has been spatiotemporally anchored. The 'complete' entities are the relevant domain for binding conditions"(ibid.3)

The grammatical function of the antecedent is part of the antecedent constraints: an anaphor must be bound or may be bound to a SUBJect. Also the domain in which an anaphor must find its antecedent is always constrained relatively to either the syntactic predicate of the the anaphoric element is an argument, the minimal domain with a subject containing the anaphor, or the minimal tensed domain containing the anaphor. These can be regarded as domain constraints. Moreover, we may think of two kinds of binding constraints: positive and negative constraints. In line with Binding Theory of Chomsky(1981), 'reflexive' is an element which must be bound or must have an antecedent within some syntactically definable domain. On the contrary, 'pronominal' is an element that must be free, or be noncoreferent with elements in some syntactically definable domain.

However if we look at "proprio", we see that it must be bound in its minimal tensed domain, but in case no suitable antecedent is available locally, it may look outside and be assigned an antecedent or even receive arbitrary reading at certain semantic conditions, definable in terms of tense, subject, aspect. As Dalrymple suggests, there may a typology of constraints rather than a typology of anaphoric elements(ibid.,4). In previous works(Hobbs, 1978; Ingria, 1989) only syntactic constituency and c-command was considered, but recent work in linguistics has clearly proven this approach to be insufficient. In particular, both Chomsky's(1981) and Manzini's(1983) theory wrongly predict the grammaticality of sentences such as,

1) *I persuaded/told the boys$_i$ that[S1 each other's$_i$ pictures were on sale.

1i) The boys$_i$ thought that each other's pictures$_i$ were on sale.

were the reciprocal anaphor each other lacking an accessible subject in its Domain Governing Category(we will not enter into a discussion of Chomsky's binding principles nor in Manzini's

modifications - see Giorgi(1984)), its Sentence (S1) is predicted to corefer freely, hence the object NP of the matrix clause is treated as a possible antecedent on a par with the subject in 1i. Since it is wrong to say that anaphors can corefer freely, what is needed is a theory of Long Distance Anaphor, which is able to explains how the anaphor is still subject to a number of binding constraints.

Here crucially, the terms long-distance and short-distance are not used in the way in which Ingria does, and do not apply to pronouns: in particular personal pronouns, cannot be treated as long-distance anaphors(see, ibid.263) since they can pick up an antecedent in any domain whatsoever, outside their minimal domain, the clause in which they are contained - including their matrix clause and the discourse. On the contrary possessive anaphors and reflexive anaphors which count as long-distance anaphors must be bound by an antecedent before leaving their matrix clause - in other words they cannot be bound by a discourse-level antecedent. This applies to lexical personal pronouns as well as to morphologically unexpressed personal pronouns like PRO/pro which can be bound in a superordinate clause or in the discourse. However reflexives in constructions involving picture noun phrases allow non-local antecedents, and rather than being subject to syntactic constraints they seem to obey discourse constraints as Pollard and Sag(1989) discuss in their work.

In the same way it is possible to explain why in the example 2) below, with an experiencing verb, the anaphor contained in the subject NP can be bound by the object which does not c-command it, showing that this notion is not sufficient in itself to tell it apart from 3) where the same structural conditions do not apply:

2) Each other's pictures$_i$ pleased the boys$_i$.

3) *Each other's wives$_i$ murdered the men$_i$

In other words each other seems to behave like a long distance anaphor, i.e. a possessive pronoun like proprio in Italian, with some exceptions. The lack of c-command is clearly shown in case a quantifier appears as experiencer,

2i. La propria$_i$ salute preoccupa ognuno$_i$/One's health worries everyone

In the same way the two Italian anaphors *sè*, which must always be governed by a preposition and *se stesso* which can also be governed by a verb, seem to behave: *sè* is differentiated from *se stesso* by the fact that it can look for a subject in a superordinate clause and by being subject-oriented, i.e. [+SUBJECTIVE]. On the contrary se stesso can be bound also by other grammatical functions and is strictly local. *Proprio*, being a mixture of both, can be bound by other grammatical functions besides the subject, and can look for a binder in a superordinate clause.

In addition, with psychic and experiencing verbs the anaphor contained in the theme/subject can be bound by the experiencer/object - the same does not apply to the pair agent/subject & theme/object of transitive verbs. In other words, candidates for antecedenthood must be selected in accordance with their status as grammatical function and thematic role. The same applies whenever the experiencer is the subject of raising verbs - when better antecedents lack - like seem/sembrare.

12)a. ?*La propria$_i$ salute preoccupa Marco$_i$/self's health worries Mark

b. La propria$_i$ salute preoccupa ognuno$_i$/self's health worries everybody

c. La malattia della propria$_i$ moglie preoccupa molto Marco$_i$/The illness of self's wife worries Mark a lot

d. *La propria$_i$ moglie odia Gino$_i$/ *La figlia della propria$_i$ moglie odia Gino$_i$

13)a. His$_i$ wife hates John$_i$

b. *His father$_i$ hates/worries everybody$_i$

As these examples clearly show, quantifier status is a very important parameter to assess the status of candidates for antecedenthood. Also, language dependent differences are clearly visible from the paradigm: Italian possesses a wider range of pronominals and anaphors and allows binding of a possessive within the same clause as embedding becomes more deeply embedded. However deep embedding does not rescue 12d: thematic relations are the relevant criterion in this case. In the corresponding English examples, binding is performed at reversed conditions: not by a quantifier is the only requirement.

Belletti and Rizzi(1988) propose for these kind of examples and for others that Principle A of the Binding Principles be an "anywhere" principle (ibid.,314), in the sense that it can apply at D-structure, where the subject NP is contained withing the VP, thus justifying the fact that the anaphor contained in the Subject is bound before it moves to its S-structure position. Obviously, this is also relevant for sentences like

14) Which picture of himself$_i$ do you think [that Bill$_i$ likes e best]?

where Move-a has destroyed the well-formed binding configuration by extracting (the constituent containing) an anaphor from the c-domain of its antecedent. In a framework like LFG, however, no such "anywhere" principle could be made to work since categories which must be bound are only visible at one level of representation. In particular, syntactic variable are visible at c-structure and this is where they must be bound by their controller; lexical anaphors are only visible at f-structure where they must be given an antecedent in their nuclear f-structure. For an example like 14 above, there is a variable binding operation that takes place at c-structure level between the FOCus wh-phrase and the empty element in the embedded clause; when we get to the next levet of representation, the anaphor contained in the FOCus is part of a syntactic chain, i.e. is included in a non-argument function, the discourse function FOCUS, and is bound to an argument function the OBJECT of the predicate "LIKE" which also assigns it its theta-role. Since the argument function is the place in which the FOCus will be interpreted, they bear the same index they can be bound under f-command, as we shall see.

3. F-command, operator binding and backward pronominalization

As we said, in order to perform binding procedures, all functional structures are transferred into a tree with

arcs and nodes, where arcs contain grammatical function. Arcs also relate each function to its mother node, allowing in this way to compute all functions contained in an upper function: this is the crucial notion for the definition of f-command dominia(see Bresnan,1982).

The algorithm uses f-command rather than c-command and obviation to prevent clitics and lexical pronouns to look for antecedents in the same f-structure in which they are contained. Formally it is expressed as follows:

F-command

For any occurrences of the functions α, β in an f-structure F, α f-commands β iff α does not contain β and every f-structure of F that contains α contains β

It is worth while reminding that f-structures coincide with lexical forms, i.e. a predicate-argument structure paired with a grammatical function assignment; in other words an fname PRED whose fvalue is a lexical form. Usually clause nuclei are the domain of lexical subcategorization, in the sense that they make available to each lexical form the grammatical functions that are subcategorized by that form (see Bresnan, 1982:304). In case also nouns are subcategorized for, the same requirement of coherence and completeness may be applied. Not all nouns however take arguments(see Grimshaw, in publication). As a consequence, "...an f-structure is locally coherent iff all of the subcategorizable functions that it contains are subcategorized by its PRED; an f-structure is then (globally) coherent iff it and all of its subsidiary f-structures are locally coherent. Similarly, an f-structure is locally complete iff it contains values for all of the functions subcategorized by its PRED; and an f-structure is then (globally) complete iff it and all of its subsidiary f-structures are locally complete."(ibid,.305) In this sense f-structure is a notion absolutely parallel to Chomsky's(1986) Complete Functional Complex, with the difference that in LFG grammatical functions are all made available in the lexical form - in particular the SUBJect -, whereas in a CFC this must be stipulated.

As for obviation, it applies to big PROs, to little pros, and to lexical pronouns: it is expressed as follows and has been incorporated in our feature system:

Obviation Principle

If P is the pronominal SUBJ of an obviative clause C, and A is a potential antecedent of P and is the SUBJ of the minimal clause nucleus that properly contains C, P is or is not bound to A according to whether P is + or - U, respectively.

Two things must be noted: first, the principle predicts that disjoint reference applies only with subject and not with nonsubject antecedents in the matrix. To distinguish reflexive pronouns which are subject-bound clause internally, in a later paper(Simpson,Bresnan, 1983), the principle has been substituted by the presence of a lexical feature [+SUBJECTIVE]. However, the conditions that must be met to bind "long anaphors" - that is reflexive pronouns which can be bound from a higher clause, and not necessarily by a subject - include mode consideration [±UNREAL], as well as the notion of f-command. In particular, the f-structure which contains the Antecedent may be the same of the one containing the Pronominal, or else be the one containing it.

A more elaborate framework results from Bresnan et al.(1985) where pronouns which must obey the Coargument Disjointness Condition (i.e. they may not be bound to an argument of the same predicate) are obviative and are marked [±NUCLEAR], thus meaning that they may or may not appear in the same syntactic nucleus as their antecedent - an ADJunct is never part of the nucleus so that a pronoun is allowed,

16a. John wrapped a blanket around him.

b. John wrapped a blanket around himself.

The English reflexive pronoun "himself" is [+NUCLEAR] and must find an antecedent within the same nucleus containing the pronominal and a subjective function; while "him" is [-NUCLEAR]. The ADJunct "around himself" however lacks a subjective function and the anaphor must look for an antecedent in the closer higher domain. However, English pronoun "him" is not obviative like the corresponding Italian one,and this fact, when added to the presence of two sets of anaphoric pronominals, gives the rather different distribution in the corresponding Italian sentences:

16i. Gino$_i$ ha visto un serpente$_j$ vicino a lui$_k$/*$_i$/*$_j$ (John has seen a snake near him)

ii. Gino$_i$ ha visto un serpente$_j$ vicino a sè$_i$/*$_j$ (John has seen a snake near "sè")

iii. Gino$_i$ ha visto un serpente$_j$ vicino a se stesso$_j$/*$_i$ (John has seen a snake near himself)

Thus, the relevant domain for anaphors and pronouns contained in nominal f-structures is not the f-structures directly containing them: this is due to their functional nature and not simply to structural reasons. As to reciprocals, reflexives and possessives anaphors are all assigned SUBJECT function thus counting as possible candidates for antecedenthood: but a conflict is raised here by the referential nature of anaphors which is marked as nonreferential in their feature matrix, hence unable to become antecedents of themselves. This conflicting result works as a filter for anaphors at the structural level, erasing their ranking as candidates for antecedenthood but raising them out of their subordinate f-structure into the upper one: in this way, anaphors cannot be bound within their minimal f-domain but must be bound in the upper one, pronouns are left free to corefer.

At clause level, reflexive pronouns look for binders in the same f-structure in which they are contained:. as we said, two kinds of anaphors must be taken care of: long anaphors like "sè", and short anaphors like "se stesso". Only short anaphors can be bound by non-subjects and only long anaphors can be bound in an upper clause if no suitable binder appears in the local minimal one. The possessive anaphor "proprio" on the contrary partakes of features belonging to both short and long anaphors: it can use both a short and a long distance strategy; it is not SUBJective. We have established then that the lexical feature [-SUBJCTV] distinguishes short anaphors from long anaphors, which are marked [+SUBJCTV]. Summarizing, we have two sets of reflexive pronouns,

a. non-subjective reflexive pronouns[-SUBJCTV] "sè"

b. subjective reflexive pronouns[+SUBJCTV] "se stesso"

In addition, long distance anaphors like the possessive "proprio", non specified as to SUBJectivity, behaves both as a long and a short anaphor, according to the domain in which it can be bound, and is posaitively marked for [+pro, +ana].

3.1 Our Proposal

Our proposal takes into account the facts of Italian in particular but also those of English, Norwegian and other languages as discussed by Enç(1989) or Dalrymple(1990). Binding is expressed by coindexation of a controller α and a controllee β, just like coreference between antecedent and pronoun, in a domain F - a complex f-structure, at the following conditions:

1. β is an f-structure [+anaph] and is bound in its F-domain
2. β is an f-structure [+pron] and is not bound in its F-domain

The first part of the formulation accounts for the fact that an anaphor is in complementary distribution with a pronoun, i.e. that in the domain in which the anaphor must be bound the pronoun must be free, or not be bound. Now, the smaller domain, is an f-structure with a SUBJect, be it an open or a closed f-structure. Obviation could be used to tell pronouns or pronominals obviative in a certain domain, an obviative proposition, that is a clause nucleus; however either formulations of obviation do not account for the behaviour of NPs. No mention seems required for referential expressions at this level, where no mention is made about the antecedent.

3. F is an F-domain iff
   α f-commands β in F and I is licensed

The second part of the formulation, says that the structure in which the antecedent and the anaphor must be bound is the one containing a SUBJect function - this is derived from the licensing condition: in an NP the F containing the head, in a clause, the F containing the SUBJect of the clause, in an ADJunct the one containing the PRO, in an open function, the open function itself.

4. F-command:

A function a f-commands a function β in F iff

a. α is not contained in β, and β is not directly contained in α, β = SUBJect

b. every f-structure of F that contains α contains β

b1. β may contain α in F iff α is in a weak RD

c. a function β is directly contained in a function α if β is a subsidiary f-structure of a function α

{the subject is not accessible to itself - the remaining arguments/adjuncts of the head Noun may be bound by the subject; as well as the i-within-i reformulated}

In a., the antecedent/binder cannot be contained in the f-structure of its bindee, in other words, the relation is asymmetric; also the bindee cannot be directly contained in the f-structure of the antecedent but it must constitute a separate f-structure. This is trivial, but requires the formulation of a notion, "directly contained", which divides f-structures contained in complements and adjuncts of a head from their governors.

The b. clause only applies when the bindee is contained in the same F that contains the binder, but the binder is down in a separate f-structure which is open. However, for the licensing conditions on F given below, obliques are not regarded as possible F-domains.

5. Licensing conditions for an Indexing I of α with β

α: 1. i. must be lexically free;
ii. it is the SUBJect
iii. it is in a strong RD
iv. its Θ-role is superior in the following hierarchy:

agent > benefactive > recipient/experiencer/goal > instrumental >theme/patient > locative

(iii. differentiates between an ADJunct PP and a predicative one, in the sense that the anaphor contained in an adjunct PP is bound to the SUBJect of the higher strong RD, whereas an anaphor contained in an open PP is bound locally to the closer function).

2. otherwise,

A. a function β is free in the discourse if F is a weak RD,

B. a function β is coreferent/cospecified in the discourse if β is in a strong RD.

6. A function is lexically free iff,
- it is argumental

A function is lexically bound iff,
- it is Ø - empty, existentially bound argument
- it is an expletive (no PRED, but FORM)
- it is a quasi-argument

7. A R(eferential)-D(omain) is an f-structure specified for referential energy:

i. it is strong iff a. it is a closed function;
b. it is referentially transparent

ii. it is weak iff a. it is an open function;
b. it is referentially opaque.

iii. Referential energy :

a. for clause nuclei(where a SUBJect is obligatory) is expressed by atomic attribute-value pairs: TENSE=[±REF]{past tense individuates a specific reference time}, MODE [±REAL]{real mode is assertive and implies the truth of the proposition-at least on part of the speaker}, CLASS[±IMPLIC] {implicative verbs imply the truth of their complements and may be interpreted referentially - also factivity is included}, ASPECT [±PERF] {perfective aspect implies the existence in the world of the object predicated by the verb};

b. for NP heads of relative predicative adjuncts CARD= [±DEF/Ø], INDIV [±SPEC], [±ref].

c. transparency obtains whenever the features have positive value.

4. The algorithm for anaphoric control

Two structuress are built from the output of the grammar: annotated c-structures, i.e. a directed graph which can be traversed primarily through syntactic constituents; and a list of the functional schemata associated with semantic forms - in other words, all PRED expressions with a list of semantic attribute-value pairs, i.e. the f-structure mapped from the previous structure, where pronominal binding is computed. The algorithm applies to a completely parsed structure which is a graph translating the annotated c-

structure of LFG into the f-structure. The algorithm uses the notions of domains used in LFG as well as functional information as to the grammatical function associated with a certain constituent, and its thematic role. The definition of domains is based crucially on the notion of f-structure and governors are derived from grammatical function and thematic role, as we shall describe in details below.

When a pronoun is encountered, the algorithm moves up to the left of its minimal domain, the closest f-structure containing it and stops in the first superordinate f-structure; on the contrary, with anaphors, the search is to the left within the same f-structure containing it, unless it is contained in a SUBJect. It is worthwhile reminding that at f-structure level the VP node disappears and an OBJect NP appears at the same level of a SUBJect NP. F-structures contained in a nominal f-structure behave differently due to their grammatical function as discussed below.

In line with Bresnan et al(1985) and contrary to the proposal contained in Dalrymple(1990) we use functional features as lexically specified properties of individual anaphoric elements. These features both account for and translate lexical category, in this way directly triggering the binding algorithm that fires a certain procedure whenever a [+anaph] feature is met in the referential table associated to a certain f-structure. Features also serve to restrict the type of possible antecedents in terms of reference to the SUBJect; to set up a hierarchy for antecedenthood in which possible antecedents are ranked according to their associated grammatical function and thematic role; to unify morphological features checking for agreement in person and number, and selectional restrictions imposed by inherent semantic features; to tell apart quantifiers and quantified NPs which cannot be used as antecedents in backward pronominalization. A complete list of features is given below.

Whenever an antecedent is found - selected by the presence of the feature [+ref] - its ranking is checked as well as its features for agreement: the interaction with binding principles determines the possibility for an OBJect referential expression to act as binder of long distance anaphors. In other words, binding works by default according to the principle "bind anaphors as soon as possible". On the contrary pronominal coreference imposes the algorithm to pick up a certain referential expression as possible candidate and to reject other referential expressions owing to their ranking in the hierarchy. Only one antecedent is selected for [+ana] elements; with [+pro] more than one antecedent is selected according to the rules and to the antecedents available.

Whenever a pronoun is left unbound the algorithm adds an instruction "resolve(x)", which is used to trigger the anaphoric binding algorithm at discourse level(see Bianchi & Delmonte, 1989). The remaining pronouns and anaphors are assigned a couple of indexes: their own and the one of their antecedent and binder. Following recent work by Enç(1989) who discusses a pronominal system for natural languages made up of seven classes, we built one made up of four classes for Italian - Chomsky's system based on two classes, anaphors and pronouns is insufficient. To be added to these four classes - which include anaphors and nouns(common, proper) - there is one class for pleonastic lexically unexpressed pronouns constituted by a verbal agreement in Italian, deprived of deictic import. Pronouns can be lexically specified or not, this being expressed by a feature introduce in Bresnan(1982), [±MU] (Morphologically Unexpressed). Thus, big PRO's resulting from tense specification which can be subject to anaphoric control - in LFG PROs are structurally or lexically functionally controlled - are differentiated from little pro's by the fact that the former are marked [+ana], and the latter are marked [-ana]. These are differentiated from clitics and independent lexical pronouns by the fact of being [+MU], whereas the latter are [-MU]. Besides, clitics are marked [+ana], whereas tonic personal pronouns are [-ana]. Epithets contain a deictic or a determiner feature specification. Pronominal quantifiers are marked [+pro] [±PART]. We give below a complete classification in features of all pronominal and nominal expressions as computed by the system, as a translation of lexical category together with features from SPEC, and NUMBER.

Table 2. Classification of pronouns anaphors and referential expressions

1.PROs[+ref,+pro,+ana,-def,+MU]
2.pros[+ref,+pro,-ana,+def,+MU]
3.clitics[+ref,+pro,+ana,+def,-MU]
4.lexical pronouns[+ref,+pro,-ana,+def,-MU]
5.epithets[+ref,+pro,-ana,±def,-MU]
6.common nouns[+ref,-pro,-ana,+class,±def,±sing]
7.partitive nouns[+ref, -pro, -ana, +class, +part, ±def, ±sing]
8.proper nouns[+ref,-pro,-ana,-class,±sing]
9.quantified NPs[+ref,-pro,-ana,±def,±part,±sing]
10. pron. quantifiers[+ref,+pro,-ana,±def,±part, ±sing]
11.null det. nouns[+ref,-pro,ana, +class, 0def, ±sing]
12. long anaphors [-ref,+pron,+ana,+SUBJCTV]
13. short anaphors [-ref,-pron,+ana, -SUBJCTV]

Other features will be attributed to nouns by their determiner: in particular articles are translated into [±DEF], numbers into [±CARD], quantifiers into [±PART]. The lack of determiner or the null determiner is marked by the presence of the feature [0 DEF]. The feature [±PART] is also assigned when a prepositional marker "di" is used to indicate an indefinite or a definite unspecified quantity (corresponding to the English "some, a (little) bit of". This information is recorded under a different functional node, the one named SPECifier, and are listed here only for convenience.

In addition, common nouns are differentiated from proper nouns by the feature +CLASS for the former and -CLASS for the latter, indicating that common nouns are used to denote classes or properties of individuals, as opposed to proper nouns which should pick out individuals. Moreover, common nouns are specified in reference by definiteness, whereas proper nouns use definiteness only redundantly - in Italian a proper noun may be preceded by a definite article. When a noun is recognized as proper, this feature is discarded. Proper nouns are assigned a higher score than common nouns, as candidates for antecedenthood. Cardinality is marked

by Number, which adds the information that a Singular, Definite, Specific noun phrase is to be interpreted as a unary set of the class of objects or individuals denoted by the noun, i.e. there is only one member referred to by the noun phrase in universe of discourse that we want to pick up. Plural noun phrases are treated differently, i.e. as quantified NPs.

5. The Basic Algorithm

We list here below the basic algorithm in its Prolog formulation: as we said previous it applies on f-structures which are compiled as a directed graph, and accessed by an algorithm with performs graph search. The complete algorithm is made up of about 4000 lines of program in Prolog.

F-structure

```
f_structure(Index,F_R,Node) :-
        node(Node):F_R:index:Index.
```

F-command

```
f_command(Alpha,Alpha_Funct,Beta,Level) :-
      f-structure(Beta,F,N), F=subj/_,
      node(N1):F1:node(N), F1 = subj/_,
      node(N2):F2:node(N1),
       f_c(N2,F2,Alpha,Alpha_Funct,O,Level_x),
       Level is Level_x + 2.
f_command(Alpha,Alpha_Funct,Beta,Level) :-
      f-structure(Beta,F,N), F=subj/_,
      node(N1):F1:node(N), F1 \ subj/_,
      f_c(N1,F1,Alpha,Alpha_Funct,O,Level_x),
      Level is Level_x + 1.
f_command(Alpha,Alpha_Funct,Beta,Level) :-
      f-structure(Beta,F,N),
       F1 \= subj/_,
      f_c(N,F,Alpha,Alpha_Funct,O,Level_x).
f_c(N,F,Alpha,Alpha_Funct,0,0) :-
                    node(N):Alpha_Funct:index:Alpha,
                     Alpha_Funct \= F.
f_c(N,F,Alpha,Alpha_Funct,Lev,Lev) :-  Lev > 0,
                    node(N):Alpha_Funct:index:Alpha.
f_c(N,F,Alpha,Alpha_Funct,Lev,Level):-
   node(N1):F1:node(N),Lev1 is Lev + 1,
f_c(N1,F1,Alpha,Alpha_Funct,Lev1,Level).
```

And this is how the main algorithm is triggered by the presence of a certain feature in the referential table associated to a certain f-structure node:

```
resolve_anaphoric(Net,Index,WeightedList) :-
                node(Node):index:Index,
                 node(Node):ref_tab:List,
                  member(+ana,List),
bagof(Outref,refer(Node,List,Outref),Listref),
                  maplist(scoring,Listref,WeightedList).
resolve_pronoun(Net,Index,WeightedList) :-
                node(Node):index:Index,
                 node(Node):ref_tab:List,
                  member(+pro,List),
bagof(Outref,refer(Node,List,Outref),Listref),
                  maplist(scoring,Listref,WeightedList).
```

Now, consider how "se stesso" is bound:

```
refer(Node,[-ref,-pro,+ana,+me],Ante/N) :-
                node(Node):index:Ind,
                 f-command(Ante,F_ante,Ind,N),N = 0,
                  F_ante = subj/_,
                  !.
refer(Node,[-ref,-pro,+ana,+me],Ante/N) :-
                node(Node):index:Ind,
                 f-command(Ante,F_ante,Ind,N),N = 1.
```

Two examples are shown here: the first is a simple case of a possessive anaphor contained in a SUBJect NP of a psychic verb: f-command is used to raise the "proprio" out of the SUBJect f-structure and the presence of an OBJect Experiences triggers binding. In the second example the long-distance anaphor "proprio" is contained in the SUBJect NP of a sentential complement: only the SUBJect of the higher clause is chosen as antecedent; the nuclear NP OBJect is discarded from the list of possible candidates because it is an Unaffected Theme (in case it were an Experiencer it would have been included).

EXAMPLE 1. La salute della propria moglie preoccupa Mario (the health of "propria" wife worries Mario)

f-structure

```
Net  ex33
index:f2
pred:preoccupare
mode:ind
tense:simple/pres
sem_cat:psych/emot
subj/causer_emot:ref_tab:[+ref,-pro,-ana,+class]
        index:np34
        pred:salute
        sem_cat:state
        gen:fem
        num:sing
        spec:def:+
        subj/posses:ref_tab:[+ref,-pro,-ana,+class]
                index:np35
                 pred:moglie
                 sem_cat:human
                 gen:fem
                 num:sing
                 spec:def:+
               subj/posses:ref_tab:[-ref,+pro,+ana,-mu]
                           index:np36
                           pred:proprio
                           gen:fem
                           num:sing
obj/experiencer:ref_tab:[+ref,-pro,-ana,-class]
        index:np37
        pred:mario
        sem_cat:human
        gen:mas
        num:sing
       spec:def:0
```

OUTPUT OF THE ANAPHORIC BINDER

```
Net index: ex33
TO RESOLVE: np36
CONTROLLED: nil
PRONOMINALS: np36[-ref,+pro,+ana,-mu]
F-COMMAND: np37/2
Possible antecedent/s of np36: [np37/101]
```

EXAMPLE 2: lui ritiene che la propria sorella ami Gino (he believes that "propria" sister loves John)
f-structure
Net ex42
index:f2
pred:ritenere
mode:indic
tense:simple/pres
sem_cat:attitude
subj/agent:ref_tab:[+ref,+pro,-ana,-mu]
index:np4
pred:lui
sem_cat:human
pers:3
gen:mas
num:sing
case:[nom]
spec: def:+
obj/prop:index:f4
pred:amare
mode:subjunct
tense:simple/pres
sem_cat:state/emot
subj/experiencer:ref_tab:[+ref,-pro,-ana,+class]
index:npl l
pred:sorella
sem_cat:human
gen:fem
num:sing
spec:def:+
subj/posses:ref_tab:[-ref,+pro,+ana,-mu]
index:np12
pred:proprio
gen:fem
num:sing
obj/theme_unaff:ref_tab:[+ref,-pro,-ana,-class]
index:np13
pred:gino
sem_cat:human
gen:mas
num:sing
spec:def:0
OUPUT OF THE ANAPHORIC BINDER
Net index: ex42
TO RESOLVE: np12,np4
CONTROLLED: nil
PRONOMINALS:[np4/[+ref,+pro,-ana,-mu],np12/[-ref,+pro,+ana,-mu]]
EXTERNAL(ex42,np4)
Possible Antecedent/s of np4: none
Possible Antecedent/s of np12: [np4/30]

6. More complex structures

6.1 Assigning Antecedents to Obviative Pronouns

Obviative pronouns in Italian can be subdivided into three different kinds: clitics, null Subject pronoun, lexical pronouns. Clitics are to be differentiated from lexical pronouns by two basic properties: they are unstressed and they can be bound in the syntax by a TOPic function. In case they are unbound at c-structure, they can be assigned an antecedent at f-structure. Lexical pronouns are always stressed, and can never be long-distance bound in the syntax. However, they can be used in doubling a local NP, as follows,

20) Il presidente ha promosso un candidato che lui, da semplice commissario, aveva bocciato.

/ The president passed a candidate which he, as a mere commissioner, had failed.

Lexical pronouns can also be used accross sentences or within the text, for contrastive or emphatic aims(see Bresnan & Mchombo(1987) on Chichewa). Finally, the Null Subject is lexically empty and behaves very closely to clitic pronouns: it can be bound in the syntax or be unbound and be assigned an antecedent at f-structure. Obviously, it cannot be stressed nor be used for emphatic, contrastive use nor for doubling. Being lexically empty makes it somewhat different from clitics in relation to the binding domain: it can be bound from within a complement clause or an adjunct clause by a lexical pronoun, but not by a common or proper Noun.

21) a. pro Ha detto che lui non verrà. / pro said that he will not come.

b. pro Ha detto che Mario non verrà.

c. pro Ha parlato di guerra perché lui ama le armi. / He has told about war because he likes weapons.

d. pro Ha parlato di guerra perché Mario ama le armi.

Only the a.- c. examples allow for coreferentiality between little pro and the lexical pronoun in the COMP - the lexical pronoun being also free to look for an external antecedent in the discourse. The same would happen in case a clitic was introduced in place of the lexical pronoun,

22) pro Ha parlato di guerra perché Mario lo conosce. / He told about war because Mario knows him.

If we front the adjunct clause, both the lexical pronoun and the clitic are available as antecedents of little pro; and also the common or proper Noun is available, since it f-commands it. However, the lexical pronoun is only available if a list of referents is intended and not to continue the discourse topic.

22) a. Poiché pro ama le armi, lui ha parlato di guerra.

b. Poiché pro ama le armi, la polizia lo controlla. / Since pro loves weapons, the police controls him.

c. Poiché pro ama le armi, Mario ha parlato di guerra.

It is a well known fact that adjunct clauses can be attached to a lower level, within a complement clause or they can be fronted therein, as in the following examples:

23) a. Gino ha detto che Maria verrà all'incontro dopo PRO aver parlato a Tom. / John said that Mary will come to the meeting after having talked to Tom.

b. Dopo PRO aver parlato a Tom, Gino ha detto che Maria verrà all'incontro. / After having talked to Tom, John said that Mary will come to the meeting.

The difference between a. and b. lies both in semantic interpretation and in the availability of antecedents for big PRO. As to semantic interpretation, the adjunct clause modifies the complement predicate in the a. example, and the matrix predicate in the b. example. As to binding of big PRO Mary will be the antecedent in a. example and John in the b. example. The skeletal f-structures for the two examples captures the different behaviour of f-command in a straightforward way:

23a. SUBJECT: Pred: Gino
PRED: DIRE <SUBJ, COMP>
SCOMP: Pred: VENIRE <OBJ> SUBJ
OBJ: Pred: Maria
SUBJ: expletive pro
ADJUNCT: Pred: Dopo
SCOMP: Pred: PARLARE <SUBJ,OBLgoal>
SUBJ: PRO
OBL: Pred: Tom

23b. ADJUNCT: Pred: Dopo
SCOMP: Pred: PARLARE <SUBJ,OBLgoal>
SUBJ: PRO
OBL: Pred: Tom
SUBJECT: Pred: Gino
PRED: DIRE <SUBJ, COMP>
SCOMP: Pred: VENIRE <OBJ> SUBJ
OBJ: Pred: Maria
SUBJ: expletive pro

In the a. example only Mary can be reached by f-command from the position of big PRO; in the b. example on the contrary, only John can be reached. The same behaviour can be predicted for little pro in tensed clauses. However, note the contrast with corresponding English complex sentences:

24) a. John beats her because he hates Mary
b. Gino la picchia perché egli/pro odia Maria
c. Gino la picchia perché Maria odia il gatto / John beats her because Mary hates the cat

As usual we indicate with italics purported coreference between the two items; now, whereas in the English example coreference between her in the matrix and Mary in the subordinate is possible, no such thing may apply to the corresponding Italian version, the b. example. Only the c. example allows it because the NP coreferent with the clitic pronoun is a SUBJect. Now, why the SUBJect should be privileged over the OBJect NP as possible antecedent for pronouns contained in a preposed subordinate clause? This is only explained in a theory of anaphora in discourse, and in particular by the fact that SUBJects are naturally used as topic of discourse or else some non canonical constituent order must be introduced in the sentence. For instance, in

25) a. Dopo che pro è arrivato, Maria ha sgridato Franco / After pro arrived, Mary scolded Frank
b. Dopo che pro è arrivato, è stato sgridato Franco
c. Dopo che pro è arrivato, Maria lo ha sgridato

coreference for little pro is only allowed in c.: the passive form with a postposed SUBJect does not permit the NP to be used as coreference, being computed as a FOCus. Being a FOCus requires a new topic of discourse to be set up and the previous references to be discarded.This is clearly shown by the specular structure in,

26) a. Dopo che è arrivato Gino, pro si è seduto. / After has arrived John, self sat down.
b. Dopo che Gino è arrivato, pro si è seduto. / After John has arrived, self sat down.
c. Dopo che pro è arrivato, Gino si è seduto. / After pro has arrived, John sat down.

where coreference in a. between Gino and pro is blocked because Gino is a focussed constituent and ARRIVARE has a lexical form with a focussed OBJect at lexical level(see Bresnan and Kanerva). When the OBJect/Theme is used as a SUBJect/Theme, however, coreference between the proper noun and the pro is possible, as shown by b.; the same applies to pro in the preposed adjunct clause and the proper noun as SUBJect of the main clause.

In order to cope with these facts, the algorithm must compute Obviation and from the obviative clausal structure see whether it can access another clausal structure at the same level or at a level below the one in which it is contained. This is done in our parser by a special procedure called "contains",

```
contains (index1,index2) :-
      node(node1):index:index1,
      node(node1):path(Bo):index:index2,
      node(node2):index:index2.
contains(index1,index2) :-
      node(node1):index:index1,
      node(node1):path(Bo):index2,
      node(node2):index:index2.
```

Here below we list the program predicate which takes care of little pros and possible antecedents contained in another clause:

```
refer(Net,Ind,[+ref,+pro,-ana,-me],Ante/N):-
            node(node):index:Ind,
            node(node):cat:features,
            node(node):num:number,
            find_gender(node,Gen),
            f_command(NAnte,F_ante,Ind,N),N > 0,
            f_structure(NAnte,F_ante,N_ante),
            not contains(NAnte,Ind),
            node(N_ante):F_sup:node(N2),
            node(N2):F/R:index:Ante,
            not node(N2):path(_):Ind,
            write(Ante/N),nl,
            node(N2):F/R:cat:Cat,
            features(Cat,features),
            node(N2):F/R:gen:Gen_ante,
    ((Gen_ante = Gen) ; (Gen = nil) ; (Gen_ante = nil)),
            node(N2):F/R:num:Num_Ante,
            number = Num_Ante,
            node(N2):F/R:ref_tab:List,
            poss_ante(Ind,Ante,List),
            non_referred_in(Ind,Ante).
```

6.2 Arbitrary or Generic Reading

All [+ana] marked pronouns do not possess intrinsic reference, being also marked [-ref] and two consequences ensue: they must be bound in their sentence and cannot look for antecedents in the discourse, unless there are additional conditions intervening, i.e. tense must be specific and not generic, and so on; they can be assigned ARBITRARY interpretation, when a controller is lacking, and a series of semantic conditions are met as to tense specification. Since ARBITRARY interpretation is a generic quantification on events this can be produced with untensed propositions or tensed ones, but with no deictic or definite import as shown by:

20)a. I think that [prop[+arbitrary]killing onself is foolish]
b. I think that [prop[+definite]killing onself has been foolish

Possessives pronouns are obviative according to whether they are contained in a predicative or open

function. A further argument may be raised for Arbitrary PROs which in LFG are introduced each time the clause does not contain a controller because being a closed function it does not need one: we quote here Bresnan(1982,345) example, in Italian,

24) E' difficile andarsene./It is difficult to leave

where the infinitive "to leave" may be analysed as an extraposed COMP bound to the SUBJect. The PRO generated as SUBJect of the predicate "LEAVE" receives [arbitrary] interpretation. In general, reflexive pronouns lacking the ability to refer independently receive their reference from their binders: in case no binder is available reflexive pronouns are assigned arbitrary or generic reference. This may be detected both from structural cues and from properties associated with the predicate of the matrix clause. In 24 the copulative sentence is a typical case in question: the adjective "difficult" may or may not select a binder for the infinitive which should appear with the preposition "for", thus turning the PRO from arbitrary to controlled,

24i. E' difficile per Gino andarsene/It is difficult for John to leave.

A similar case may be raised for anaphoric pronouns, whenever they are contained in a subject NP, as follows,

25) La propria$_{arb}$ libertà è una cosa importante/One's freedom is an important thing

The sentence contains a generic statement absolutely parallel to the reading of 24; the same happens whenever the anaphoric pronoun is contained in the subject position of a closed function like a sentential complement,

26) Marta$_i$ pensa che la propria$_{i/arb}$ libertà sia una cosa importante/ Martha thinks that one's freedom be an important thing

in a parallel way to the behaviour of PRO

26i) Mary thinks that [PRO to behave oneself is important.

We may note at this point the fact that English possessive pronouns behave in a different way from Italian ones: in particular "his" may be bound by a quantifier through PRO, and it may be taken to corefer to a non c-commanding NP, differently from what happens in Italian,

27) *La sua$_i$ salute preoccupa ognuno$_i$

28) PRO Knowing his$_i$ father pleases every boy$_i$ ≠ Conoscere proprioi/suox padre fa piacere a ognii ragazzo

29) His$_i$ mother loves John$_i$ ≠ Sua$_x$ madre ama Gino$_i$

In particular, "his" seems to possess the ability to be bound by quantifiers like "proprio" does: in 28 the Italian version becomes analogous to the English one if we substitute "proprio" to "suo". In other words, Italian has two separate lexical pronouns for bound and unbound reference whereas English has only one and the conditions on binding are simply structural whereas in Italian they are both structural and lexical.The peculiarity of long-distance anaphors emerges from the dependency of binding on the presence of a feature at sentence level, the one related to the mood of the subordinate clause. In particular, as also detected in other languages (cf. Zaenen, 1983) the choice of Indicative vs. Subjunctive Mood is relevant for the binding possibilities of anaphors contained in the clause. The presence of the Indicative, in the most embedded clause, the one containing the long-distance anaphor seems to block binding from the matrix clause, as shown in:

30) Gino$_i$ pensa che tu sia convinto che la propria$_i$/*$_{arb}$ famiglia sia la cosa più importante.

31) Gino$_i$ pensa che tu sei convinto che la propria*i/$_{arb}$ famiglia è la cosa più importante. /John thinks that you be/are convinced that self's family be/is the most important thing.

where we changed subjunctive in 30 to indicative in 31: only 30 allows binding, hence bound reference, and disallows arbitrary reference; on the contrary 31 only allows arbitrary reference i.e. no reference at all. As discussed at length in Zaenen(1983) the choice of the mood is bound by the matrix verb which permits only certain kind of referential acts to be realized by the complement clause. Being lexical, this information can be easily transmitted in features to the c-structure and percolated according to the usual LFG conventions(see Giorgi,1984, for a lexical typology of the governing verbs).

The same applies to derived nominals like "suspicion" which can be the head of a sentential complement, inducing long-distance binding or preventing according to the presence of [+BOUND] feature,

32) Gino$_i$ ritiene che il sospetto di Carlo$_j$ che la propria$_{i/j}$ sorella sia un assassino abbia determinato la sua condanna.

33) Gino$_i$ ritiene che l'affermazione di Carlo$_j$ che la propria*i/j sorella è un assassino abbia determinato la sua condanna.

/ John believes that the Karl's suspicion that self's sister be/is a murdered had determined his/her trial.

6.3 Quantifiers and quantified NP's as antecedents

As a first approach to the problem of quantifiers, the algorithm takes care of precedence whenever a quantifed NP is indicated as possible antecedent for a pronoun. Quantified antecedents are individuated by the presence of the feature ±part in SPEC, as follows,

```
34) quantified(Ante) :- node(N):index:Ante,
                        node(N):spec:part:_.
```

This predicate is used for quantified antecedents in a simple declarative with psychic verbs: as discussed above, binding of a possessive long distance anaphor can take place from a quantified antecedent contained at clause level.

However, when we want to deal with quantifiers and quantified NPs as possible antecedents of little pros, clitics or independent pronouns a different procedure must be called in, and is the following one,

```
35) a. non_quantif(Ante) :- node(N):index:Ante,
                            not node(N):spec:part:_, !.
    b. non_quantif(Ante) :- node(N):index:Ante,
                            node(N):spec:part:X,
                            (X = '-'),
                            node(N):spec:def:'+'.
```

This procedure is integrated into the predicate for referring clitics, in particular as follows,

```
36) refer(Net,Ind,[+ref,+pro,+ana,+me],Ante/N):-
```

```
node(node):index:Ind,
node(node):cat:features,
node(node):num:number,
node(node):gen:gender,
find_gender(node,Gen),
f_command(NAnte,F_ante,Ind,N),N > 0,
f_structure(NAnte,F_ante,N_ante),
not contains(NAnte,Ind),
node(N_ante):F_sup:node(N2),
node(N2):F/R:index:Ante,
non_quantif(Ante),
not node(N2):path(_):Ind,
node(N2):F/R:cat:Cat,
features(Cat,features),
node(N2):F/R:gen:Gen_ante,
node(N2):F/R:num:Num_Ante,
number = Num_Ante,
node(N2):F/R:ref_tab:List,
poss_ante(Ind,Ante,List),
non_referred_in(Ind,Ante).
```

In this way we can account for lack of coreference between a clitic pronoun contained in a fronted subordinate clause and a quantified NP contained in the main clause, as in the a. example

37)a. When I insulted him, every student went out of the room.

b. When I insulted him, John went out of the room.

as opposed to the b. example, where coreference is allowed as usual. Here below we show the f-structure and the anaphoric binding processing results of the two sentences:

```
Net ex28
index: f1
main: index:f5
      pred:go_out
      mood:indic
      tense:past/simple
      cat:extensional
      aspect:accomplishment
      subj/agent:ref_tab:[+ref,-pro,-ana,+class]
                  index:np6
                  pred:student
                  gen:mas
                  num:sing
                  pers:3rd
                  spec:def:0
                        part:-
                        quant:every
      oblique/locative:ref_tab:[+ref,-pro,-ana,+class]
                  index:np7
                  pred:room
                  gen:mas
                  num:sing
                  pers:third
                  spec:def:+
adj:pred:when
   subordinate_clause:index:f3
      pred:insult
      mood:indic
      tense:past/simple
      cat:evaluative
      aspect:achievement
      subj/agent:ref_tab:[+ref,+pro,-ana,+me]
                  index:np4
                  pred:I
                  gen:nonspec
                  num:sing
                  pers:first
                  spec:def:+
      obj/theme_affect:ref_tab:[+ref,+pro,-ana,+me]
                  index:np5
                  pred:him
                  gen:mas
                  num:sing
                  pers:first
                  case:acc
                  spec:def:+
OUPUT OF THE ANAPHORIC BINDER
Net index: ex28
TO RESOLVE: np5
CONTROLLED: nil
PRONOMINALS:[np5/[+ref,+pro,-ana,-mu]]
EXTERNAL(ex28,np4)
Possible Antecedent/s of np4: none
Net ex29
index: f1
main: index:f5
      pred:go_out
      mood:indic
      tense:past/simple
      cat:extensional
      aspect:accomplishment
      subj/agent:ref_tab:[+ref,-pro,-ana,-class]
                  index:np6
                  pred:John
                  gen:mas
                  num:sing
                  pers:3rd
                  spec:def:+
      oblique/locative:ref_tab:[+ref,-pro,-ana,+class]
                  index:np7
                  pred:room
                  gen:mas
                  num:sing
                  pers:third
                  spec:def:+
adj:pred:when
   subordinate_clause:index:f3
      pred:insult
      mood:indic
      tense:past/simple
      cat:evaluative
      aspect:achievement
      subj/agent:ref_tab:[+ref,+pro,-ana,+me]
                  index:np4
                  pred:I
                  gen:nonspec
                  num:sing
                  pers:first
                  spec:def:+
      obj/theme_affect:ref_tab:[+ref,+pro,-ana,+me]
                  index:np5
                  pred:him
                  gen:mas
                  num:sing
                  pers:first
```

case:acc
spec:def:+
OUPUT OF THE ANAPHORIC BINDER
Net index: ex29
TO RESOLVE: np5
CONTROLLED: nil
PRONOMINALS:[np5/[+ref,+pro,-ana,-mu]]
EXTERNAL(ex29,np4)
Possible Antecedent/s of np4: [np6/131]

This notion of binding relevant for long-distance anaphors is also important for quantifiers as discussed in another work(Delmonte, 1989), in particular the fact that pronouns embedded in an Indicative or [-BOUND] clause need referential antecedents and not arbitrary or generic ones, as shown by the pair

34) A woman requires/demands that many/every men be in love with her, *and John knows her.

35) A woman believes that many men like her, and John knows her.

in 34, in English as in Italian, the indefinite "a woman" is computed as generic in the main clause and the same happens to the pronoun "her" in the complement clause introduced by "that"; but the conjoined sentence is expressed in the indicative and requires a specific woman to be picked up for referring the pronoun "her", which in this case must be computed as referential and not as generic, so the sentence is ungrammatical. The opposite happens in 35, where the indefinite is taken to refer to a specific woman in the discourse, and the two occurrence of "her" to be bound to this individual. As clearly shown, the referential capabilities of pronouns are tightly linked to the ones of their antecedent: but the opposite may happen, i.e. the referential abilities of the antecedents are bound by those of the pronouns, and these in turn are conditioned by the referential nature of the RD- referential domain - in which they are contained: an [-BOUND] domain is one containing indicative mood and reference is free, whereas a [+BOUND] domain is one containing a subjunctive mood and reference not free but locally bound, for anaphors, or lacking in referential import for lexical pronouns.

7. Chains and Binding

As we know, when at c-structure level a syntactic variable is bound to a TOPic or a FOCus a chain is created, which essentially is a couple of f-structures carrying the same index. One of the two members of the chain - the tail, is the controlled or bound element: this is an argument function and carries a theta-role; on the contrary, the head of the chain, the controller or binder is a non-argument function and has no theta-role. At f-structure level, the chain counts as a single element, in other words, the head of the chain plays no independent referential role from its tail, which is the argument function. Thus a short anaphor can be bound by the tail of a syntactic chain if contained in the same clause. On the contrary the head of the chain, which is contained in the higher domain cannot be the antecedent of anaphors or pronouns. The head of the chain, in turn, can contain a referring expression, a quantified expression, a pronoun or an anaphor: in the latter case, the tail cannot act as an antecedent, being conindexed with an element which must be itself bound in some domain. The domain is the one of the tail to which the anaphor contained in the head of the chain must be bound. We shall discuss some examples, now:

36) a. Parlando di suo suocero, Nixon ha ordinato a Bush, che lo ascoltava, di lasciarlo perdere.
/ Talking about his brother-in-law, Nixon ordered Bush, who listened to him, to let him go.

b. A se stesso Franco crede che Tom non pensa e mai.
/ Himself Frank believes that Tom never thinks to.

c. Parlando di se stesso, Nixon ha detto a Bush che ama la propria famiglia. / Talking about himself, Nixon told Bush that he loves his own family.

Consider a. and the status of suo/his: it is contained in an OBLique/theme and as such it can either be bound to the local SUBJect, big PRO, which in turn being contained in an untensed adjunct is bound under f-command by the SUBJects of the matrix, or be free and be bound to the coargument of the matrix SUBJect, the OBJ2 "Bush". Now consider lo/him which is contained within the non-restrictive relative clause: being a pronoun it is obviative within its minimal clause and must look in the higher f-structure, the matrix clause. At this level, two possible antecedents seem to be available: Nixon and Bush. However, Bush is already bound to the relative pronoun which is the SUBJect of the relative clause that contains the pronoun lo. Thus, it must be eliminated from the list of the possible candidates. In example b. a short anaphor se stessa/herself has been left dislocated and is thus bound to its bindee in the embedded clause: since the anaphor requires a binder, and the interpretation of the anaphor is derived from the location of its bindee, the antecedent of the anaphor should be found in its minimal clause. Tom is thus the binder of the anaphor and not Frank.

Finally, in the c. example, the anaphor contained in the adjunct clause is bound only to big PRO and this in turn is anaphorically controlled by the SUBJect of the matrix, Nixon. Differently from the pronoun in the a. example, the anaphor cannot pick Bush as its possible antecedent. Now consider propria/his own: the reportive verb of the matrix dire/say requires the matrix SUBJect to bind the lower little pro and thus to act as antecedent for the possessive anaphor.

The main predicate which spots chain members contained in a separate f-structure from the one containing the variable and the reflexive or pronominal element is non_referred_in, which we list here below:

```
non_referred_in(index,Ante) :-
                    pair_level(index,ListPair),
                    maplist(find_ind,ListPair,ListInd),
                   not referenced(Ante,[],ListInd).
referenced(N,Path,ListPair) :-
                     member(N,ListPair), !.
referenced(Npx,Path,ListPair) :-
       (antecedent(_,Npx,Np1);antecedent(_,Np1,Npx);
        controlled(Npx,Np1);controlled(Np1,Npx)),
         not member(Np1,Path),
         riferimento(Np1,[Npx|Path],ListPair).
find_ind(node/_,Ind):- node(node):index:Ind, !.
find_ind(node/_,nil).
```

This predicate deletes from the list of possible antecedents for lexical pronouns the Np head of the

chain, and takes as local binder of a reflexive the controlled variable or tail of a chain.
Let's consider now more closely the English version for 36b., with examples taken from Barrs(1988). First of all, the English version which we repeat here below, where we indicate with superscripts the syntactic index and with subscript the anaphoric index,

36b. Himself$_{j/k}$, Frank believes that Tom$_k$ never thinks to e$_j$.

has a lexical anaphor "himself" which can be bound both by Frank and by Tom. This is not allowed in Italian: in other words, Italian requires the anaphor to be "reconstructed" back into the place from which it has been extracted to produce the Topicalized structure. This is possible by considering the variable as the tail of a chain and the topicalized element as its head. Barrs's examples are very similar (his 7a,42)

37)a. Which pictures of himself did John say Bob liked e?
b. Himself, he thinks Mary loves e.

in 37a, the sentence is ambiguous - either John or Bob may be interpreted as the antecedent of the reflexive, in the b. example binding by "he" is grammatical, however in the corresponding Italian examples, no such ambiguity may arise and the b. version becomes ungrammatical,

38)a. Quali foto di se stesso Gino ha detto che Bruno ama e?
b. *Se stesso, egli pensa che Maria ama e.

Ungrammaticality is readily explained by the fact that "se stesso" must be locally bound and "Maria" is not an adequate SUBJect binder because of failure of agreement features. Two cases suspend ambiguity: the anaphor is contained in a predicative function, an ACOMP, or there is an accessible SUBJect, and are illustrated by the following examples, (his 7b, 17)

39)a. Whose pictures of himself did John say Bob liked e?
b. How proud of himself did John say Bob became e?

In 39a. the possessive pronoun "whose" provides a POSSessor or a SUBJect for the binding of the anaphor in its minimal local domain; in 39b. the head predicate "proud" is a predicative function with a functionally controlled SUBJect which is lexically bound to the available SUBJect "Bob". This happens before f-structure is accessed, so that no more binding domains may be accessed. Barrs gives a version within Chomsky's(1986) "Barriers" framework and Higginbotham's(1983) Linking Theory which accounts for the same facts in a transformation model.

8. Current Status and Comparison with Related Work

In using f-structures rather than syntactic constituency, LFG makes it more natural and direct looking for information such as being the "subject of", a notion crucial for antecedenthood.
Each referring expression receives a separate treatment by the rules for binding according to its feature matrix, grammatical function, and thematic role. For instance, little pro and clitics are included in the same class, but their grammatical function is crucial for distinguishing among them in their ability to be bound by an antecedent: little pro's can only be bound by subject antecedents, or nominative ones, whereas clitics being assigned accusative, dative or oblique can never be bound by a subject antecedent.

A set of criteria for assigning priority scores to candidates for antecedenthood and binding are used in order to define what can be bound by what: candidates receive scores according to their grammatical function, SUBject scoring the highest; and to thematic role, agent scoring the highest, and so on. Exceptions are also individuated on the basis of the interplay of grammatical functions and thematic roles: for instance one such rule says that a possessive anaphor contained in a subject f-structure can be bound to a NP in its sentence unless it is a Theme.As appears, binding is crucially performed on a structural basis, rather than on a functional basis as the approach based on Functional Uncertainty would require. The structures involved are f-structures: the parser makes reference to the SUBJect a primitive notion which is used primarily to set NP f-structure apart from clausal ones; untensed clauses may either appear as controlled complements, or as closed adjuncts or closed functions such as SUBJect: also in this case anaphoric binding applies as long as structural conditions allow it. In this sense, anaphoric binding together with syntactic binding are structurally determined and can be opposed to lexical binding which is entirely functionally determined. Scores are also very important and are based on the superiority hierarchy of theta-roles, and on the degree of referentiality a certain NP possesses.
In particular, the difference in binding domain existing between an anaphor like "himself" and a pronoun like "him" is obtained simply by reference to the level at which these two lexical items must start out looking for their antecedent: for the former it would be equal to 0, while for the latter would be equal to 1. Rather than formulating a "Coargument Disjointness Condition" it is sufficient to individuate a viable f-structure, which looks for the accessible SUBJect in the case of nominal ones and let the feature matrix do the rest.

As we saw, reference to the particular domain in which a certain element must be bound or be disjoint, and reference to the particular grammatical function the antecedent should bear in a particular environment is not sufficient to deal with the inventory of pronominals available in Italian and other languages: reference to the thematic role is sometimes required, whenever a psychic verb is used, as well as the type of quantified NP or quantifier that can become a candidate for antecedenthood in certain environments. Our systems does this directly by means of the feature matrix associated to the referential table and by directly investigating the content of the functional node, where theta-roles are available together with the function label. Possibly, the same result could be achieved by means of Functional Uncertainty, even though we have not tried to test this hypothesis.

However, let us consider why Functional Uncertainty has been introduced: basically because syntactic restrictions could be formulated in terms of grammatical functions, and could be expressed by the

introduction of equation whose right-hand side member contained regular expressions like the following,

(37) ($\uparrow$ TOPIC) = ($\uparrow$ COMP* OBJ)

which refers to the analysis of Topicalization as discussed by Kaplan & Zaenen(1989). The equation specifies an infinite disjunction of paths within f-structures, paths involving zero or more COMPs: OBJ stands for the landing site or for the bindee for the binder. Using functional attributes makes things easier and does completely away with the need to keep in memory c-structure syntactic trees once they have been used to build the corresponding f-structures. I don't intend here to comment on Kaplan & Zaenen proposal, but simply to criticize Dalrymple's idea to use this procedure with some minor modification and adaption in anaphoric binding.

It is clear to me that the regularity of syntactic phenomena has a different nature from the one belonging to anaphoric ones. An equation like the one reported in (37) states that no matter what happens within the COMP, and as long as the landing site is an OBJ, any number of COMP's may be traversed in order to adequately bind the TOPIC. This never happens with anaphoric binding: even though the difference existing between ADJunct clauses and COMPlement ones is relevant, the depth of embedding is also a crucial factor. Structural differences like the one existing between COMP and ADJ clauses are already taken care for by f-command: however, in order to let, say, a long-distance anaphor or a clitic pierce through, inside-out, more than one relevant domain, a number of conditions on antecedenthood and distance intervening between the anaphor and the antecedent must be also accounted for.

References

Barrs A.(1988), Paths, Connectivity, and featureless empty categories, in *Annali di Ca' Foscari* XXVII, 4, 9-34.

Belletti A., Rizzi L.(1988), Psych-Verbs and Theta-theory, *Natural Language and Linguistic Theory* 6, 3, 291-352.

Bratko I.(1986), *Prolog Programming for Artificial Intelligence*, Addison-Wesley Pub.Co.

Bresnan J.(ed)(1982), *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge Mass.

Bresnan J.; Per-K.Halvorsen; J.Maling(1985), Logophoricity and bound anaphora, MS, Stanford University.

Bresnan J., Mchombo J.M.(1987), Topic, Pronoun and Agreement in Chichewa, *Language* 63, 4, 741-782.

Chomsky N.(1986), *Barriers*, MIT Press, Cambridge, Mass.

Dahl V.(1981), Translating Spanish into Logic through Logic, *American Journal of Computational Linguistics* 7, 3, 149-164.

Dalrymple M.(1990), Syntactic Constraints on Anaphoric Binding, Ph.D. Dissertation, Stanford University.

Delmonte R.(1985), Parsing Difficulties & Phonological Processing in Italian, *Proceedings of the 2nd Conference of the European Chapter of ACL*, Geneva, 136-145.

Delmonte R.(1989), Grammatica e Quantificazione in LFG, MS, University of Venice.

Delmonte R.(to appear)(1990), Semantic Parsing with LFG and Conceptual Representations, *Computers & the Humanities*, 5-6,pp.30.

Enç M.(1989), Pronouns, Licensing, and Binding, Natural Language and Linguistic Theory 7, 1, 51-92.

Higginbotham J.(1983), LF, Binding and Nominals, *Linguistic Inquiry* 14, 395-420.

Ingria R., D.Stallard(1989), A Computational Mechanism for Pronominal Reference, in *Proceedings of the 27th Annual Meeting of ACL*, Vancouver, 262-271.

Kaplan R., A.Zaenen(1989), Long-distance dependencies, constituent structure, and functional uncertainty, in M.Baltin & A.Kroch(eds), *Alternative Conceptions of Phrase Structure*, Chicago University Press.

McKeown K., C.Paris(1987), Functional Unification Grammar Revisited, in *Proceedings of the 25th Annual Meeting of the ACL*, Stanford, 97-103.

Pereira F.(1981), Extraposition Grammars, *American Journal of Computational Linguistics* 7, 4, 243-256.

Pereira F.(1983), Logic for Natural Language Analysis, Technical Note 275, Artificial Intelligence Center, SRI International.

Pereira F.(1985), A Structure-Sharing Representation for Unification-Based Grammar Formalism, in *Proceedings of the 23rd Annual Meeting of the ACL*, Chicago, 137-144.

Pollard C., I.Sag(1989), Anaphors in English and the scope of the binding theory, MS, Stanford University.

Sells P.; A.Zaenen; D.Zec(1987), Reflexivazation variation: Relations between syntax, semantics, and lexical structure. In M.Lida; S.Wechsler; D.Zec(eds) Working Papers in *Grammatical Theory and Discourse Structure*, 169-238, CSLI/University of Chicago Press, Stanford University, CSLI Lecture Notes, N.11.

Zaenen A.(1983) On Syntactic Binding, *Linguistic Inquiry* 14, 3, 469-504.

P.S. All modules have been implemented in Prolog and run both under Macintosh, MS-Dos and VMS with Quintus Prolog.

Acknowledgements

The programming work by Paolo Frugoni is kindly acknowledged. Support for the research has been partly provided by the PROMETHEUS Project, Man-Machine Interface, PFT "Trasporti", under CNR, contract number 89.01470.93.

A HYBRID MODEL OF HUMAN SENTENCE PROCESSING: PARSING RIGHT-BRANCHING, CENTER-EMBEDDED AND CROSS-SERIAL DEPENDENCIES

THEO VOSSE
GERARD KEMPEN
N.I.C.I., University of Nijmegen
Montessorilaan 3
6525 HR Nijmegen
The Netherlands
Phone: +31 80 512621
Internet: vosse@psych.kun.nl
kempen@psych.kun.nl

ABSTRACT

A new cognitive architecture for the syntactic aspects of human sentence processing (called Unification Space) is tested against experimental data from human subjects. The data, originally collected by Bach, Brown and Marslen-Wilson (1986), concern the comprehensibility of verb dependency constructions in Dutch and German: right-branching, center-embedded, and cross-serial dependencies of one to four levels deep. A satisfactory fit is obtained between comprehensibility data and parsability scores in the model.

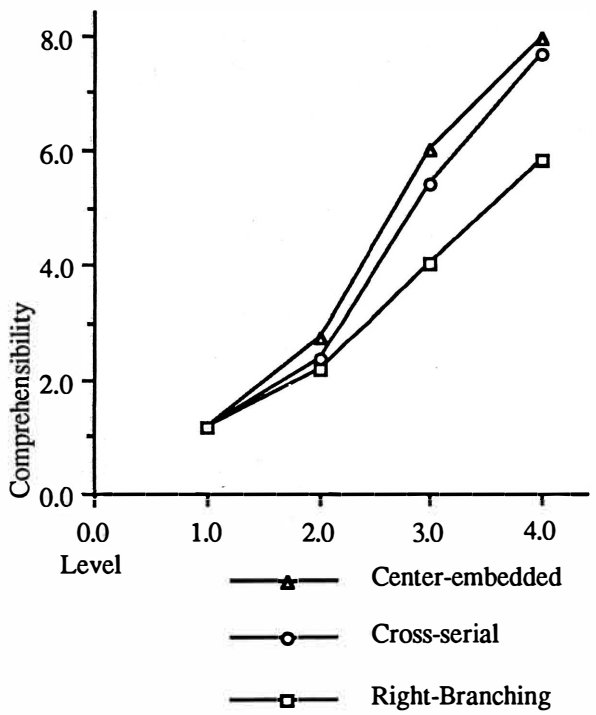


Figure 1. Comprehensibility ratings for various construction types and depths (1 = very easy, 9 = very hard).

INTRODUCTION

In a recent paper (Kempen & Vosse, 1990), we have proposed a new cognitive architecture for the syntactic aspects of human sentence processing. The model is 'hybrid' in the sense that it combines symbolic structures (parse trees) with non-symbolic processing (simulated annealing). The computer model of this architecture — called *Unification Space* — is capable of simulating well-known psycholinguistic sentence understanding phenomena such as the effects of Minimal Attachment, Right Association and Lexical Ambiguity (cf. Frazier, 1987).

In this paper we test the Unification Space architecture against a set of psycholinguistic data on the difficulty of understanding three types of verb dependency constructions of various levels of embedding[1].

[1] A recent paper by Joshi (1990) motivated us to do the present study. He succeeds in obtaining a good fit between Bach *et al.*'s data and a complexity measure deriving from his model, which is based on an Embedded Push-Down Automaton (EPDA) and Tree Adjoining Grammar (TAG).

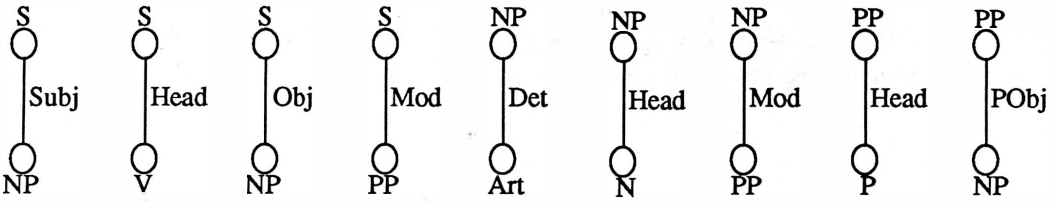


Figure 2. Various types of synactic segments.

The data were collected by Bach, Brown and Marslen-Wilson (1986) and concern comprehensibility ratings of cross-serial, center-embedded and right-branching constructions as illustrated by (1). Subjects rated two types of verb dependencies: right-branching and either center-embedded (German) or cross-serial (Dutch) dependencies.

Dependency type

(1a) ... when John saw Peter walk
Right-branching

(1b) ... als Johan Peter laufen sah
Center-embedded (nested)

(1c) ... toen Jan Peter zag lopen
Cross-serial (crossed)

The right-branching constructions are quite common in Dutch and German. German sentences were rated only by native speakers of German, Dutch sentences only by native speakers of Dutch. Figure 1 shows the obtained comprehensibility (or rather, *in*comprehensibility) ratings for four 'levels' (the term level refers to the depth of embedding; level 1: one clause, without embeddings; level 2: two clauses, one embedded in the other as in (1), etc.). Notice that the (Dutch) crossed dependencies were consistently rated easier to understand than the (German) nested dependencies. From level 3 onward, the right-branching structures were judged easier than their crossed or nested counterparts. Via a question-answering task Bach *et al.* verified that the comprehensibility ratings indeed reflect processing loads (real difficulties in comprehension).

In Section 2 we outline briefly the type of grammar we use to represent syntactic structures. The parsing mechanism capable of building such structures is described in Section 3. Section 4 is devoted to design and results of the computer simulation. In Section 5, finally, we evaluate our results and draw some comparisons with alternative computational models proposed in the psycholinguistic literature.

SEGMENT GRAMMAR

Kempen (1987) introduced Segment Grammar as a formalism for generating syntactic trees out of so-

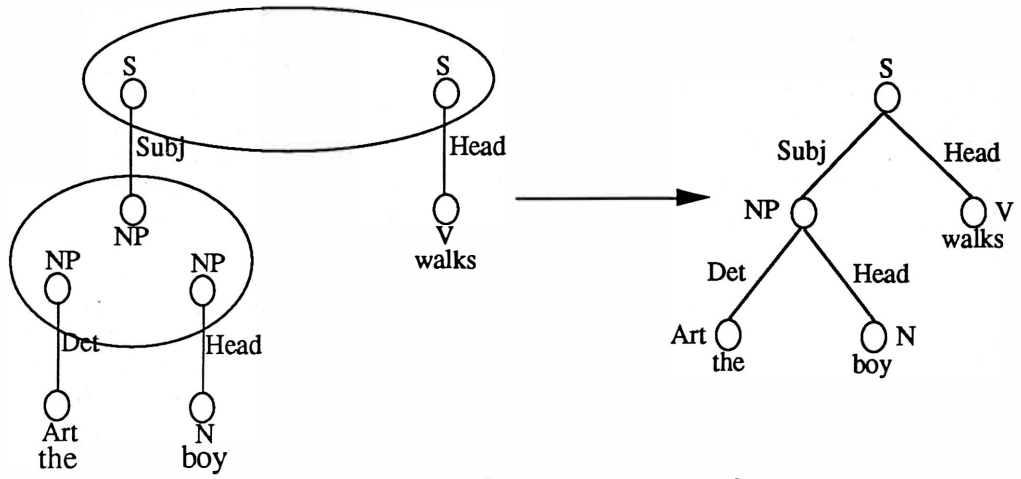


Figure 3. Building a tree through unification.

called segments. A segment is a node-arc-node triple, the top node being called 'root' and the bottom node 'foot'. Both root and foot nodes are labeled by a syntactic category (e.g. S, NP) and have associated with them a matrix of features (i.e., attribute-value pairs). Arc labels represent grammatical functions. See Figure 2 for some examples. All syntactic knowledge a segment needs (including ordering rules) is represented in features.

The basic tree formation operation is *unification* of the feature matrices of nodes which carry the same category label. In Figure 3 successful unification has been visualized as the merger of the corresponding nodes.

Segment Grammar is completely lexicalized. Every lexical entry specifies a single segment or a sub-tree consisting of several segments. For instance, one entry for the English verb *eat* looks like Figure 4. It specifies the subcategorization features for this verb, including the fact that it can take zero or more modifiers (Mod*) in the form of prepositional or adverbial phrases. For more details about Segment Grammar (including the Dutch sentence generator based on it) see De Smedt (1990).

THE UNIFICATION SPACE

The dynamics of the Unification Space model were inspired by the metaphor of bio-chemical synthesis. Think of the segments as molecules floating around in a test-tube and entering into chemical bonds with other molecules (unification of nodes). The resulting larger structure may be insufficiently stable and fall apart again. After a break-up, the segments continue their search for suitable unification partners until a stable 'conformation' — that is, the final parse tree — has been reached.

Henceforth, we denote the test-tube by the term Unification Space. Words recognized in the input string are immediately looked up in the mental lexicon and the lexical entry listed there is immediately entered into the Unification Space. In case of an ambiguous input word, all entries are fed into the system simultaneously.

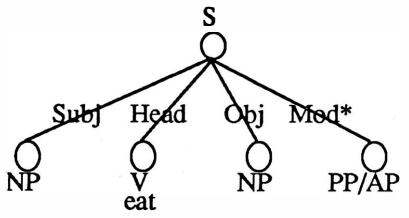


Figure 4. Lexical entry for the transitive verb 'eat'.

The following principles control the events in the Unification Space (see Kempen & Vosse, 1989, for details):

- *Activation decay.* When the nodes are entered into the Unification Space they are assigned an initial activation level by their lexicon entry. This activation level decays over time.

- *Stochastic parse tree optimization.* Generally, on the basis of its feature composition, a node could unify with several other nodes present in the Unification Space. In order to make the best possible choice, Simulated Annealing is used as a stochastic optimization technique (cf. Sampson, 1986). If two nodes *can* unify, they actually unify with probability p_U. This probability depends, among others, on the activation level of both nodes and on the grammatical 'goodness of fit'. Various syntactic and semantic factors are at stake here. Among the former are word order constraints. For instance, if during the analysis of *He gave that girl a dollar* the article *a* would attempt to unify with the noun *girl*, this would cause violation of a word order rule and drastically reduce the value of p_U. Assigning *a dollar* the role of indirect object would be evaluated as less good than as direct object, both for syntactic and semantic reasons.

  On the other hand, unified nodes may break up, with probability p_B. This probability increases accordingly as the activation of the nodes and/or their grammatical goodness of fit decrease. One consequence of this scheme is a bias in favor of semantically and syntactically well-formed syntactic trees encompassing recent nodes.

- *Global excitation.* Due to the spontaneous decay of node activation and the concomitant rising p_B, all unifications would ultimately be annulled in the absence of a mechanism for intercepting and 'freezing' high-quality parse trees. In standard versions of simulated annealing one obtains this effect by making both p_U and p_B dependent on a global 'temperature' variable T which decreases gradually according to the 'annealing schedule' which has

been determined beforehand. We define a parameter E (for global Excitation) whose function is similar to that of temperature. However, E's value does not decrease monotonically as prescribed by some annealing schedule but is proportional to the *summed activations* of all nodes that currently populate the Unification Space.

The relation between E on one hand and p_U and p_B on the other is such that, after E has fallen below a threshold value ('freezing'), no unifications are attempted anymore nor can unified nodes become dissociated. If the resulting conformation consists of exactly one tree, the parsing process is said to have succeeded. If several disconnected, partial trees result, the parsing has failed.

It is important to note that the workings of the Unification Space prevent the parallel growth of multiple parse trees spanning the same input string. In other words, *structural* (syntactic) ambiguity is not reflected by multiple parse trees. Only in case of *lexical* ambiguity can there be parallel activation of several segments or subtrees. This agrees with the picture emerging from the psycholinguistic literature (cf. the survey by Rayner & Pollatsek, 1989).

We now describe the essence of the computer implementation of the Unification Space model. Mathematical details can be found in Kempen & Vosse (1989).

1. Time is sliced up into intervals of equal duration. During each cycle, one iteration of the basic algorithm is carried out. This process stops when E has fallen below the threshold value.

2. Words recognized in the input sentence are stored in an input buffer for a limited period of time, T_B. Individual words are read out from left to right at fixed intervals $T_w << T_B$. Their corresponding lexical entries are immediately entered into the Unification Space.

3. During each cycle, two nodes, n_1 and n_2, are picked at random. If their feature composition permits unification, they actually unify with a probability of p_U which covaries with n_1's and n_2's activation levels. The activation level of the resulting single node is higher than the activation level of either n_1 or n_2.

4. Then, for each segment in the Unification Space, it is determined whether or not it will dissociate from its unification partner (if any). This event takes place with probability p_B which correlates negatively with the activation level. Whenever *lexical* segments are are involved in a break-up (lexical segments have word classes rather than phrases as their foot labels), their lexical entries are reentered into the Unification Space without delay. Thus they are given a new chance to find a suitable unification partner. The activation levels of reentering nodes are reset to the initial value stored in the lexicon. However, if a word has already been dropped from the input buffer, its lexical entry is *not* reentered.

5. The activation levels of all nodes are adjusted on the basis of the decay parameter and the new value for E is computed.

THE SIMULATION STUDY

In our earlier study we obtained satisfactory simulation results for the sentences in (2).

(2a) The rat the cat chased escaped.
(2b) The cat chased the rat that escaped.
(2c) The rat the cat the dog bit chased escaped.
(2d) The dog bit the cat that chased the rat that escaped.

The Simulation Space had virtually no problems in parsing doubly embedded sentences (2a) and (2b): the number of correct solutions was close to 100 percent. However, this score dropped considerably for triply embedded clauses: to about 80 and 50 percent for righthand and center-embeddings respectively[2]. This pattern is in good agreement with psycholinguistic observations.

In order to avoid controversial assumptions about the syntactic structure underlying cross-serial dependencies, we have devised simple artificial grammars which generate right-branching, center-embedded and cross-serial dependencies among pairs of opening and closing brackets, e.g. '0{}', '({})' or '({)}'. The grammars contain two types of lexical segments (with arc labels Left and Right) and one optional type of non-lexical segments with arc label Mod. The number of Mod segments dominated by an S node is either zero or one The optional Mod segment is attached to the lexical entries of opening brackets as depicted in Figure 5. It is the Mod segments that give the grammar a recursive flavor.

The S nodes have associated with them a 'bracket type' feature whose value is 'round', 'curly', 'square', etc. This prevents unification of S nodes

[2] These numbers have been computed as described in footnote 3 below.

that dominate brackets of different types, e.g. S-Left-{ with S-Right-].

The sole difference between the three grammars rests in their word order constraints. Center-embeddings require the embedd*ed* subtree to be positioned inbetween the branches of the embedd*ing* S. (The constraints for both other grammars are easy to devise.) However, there was no need to have the Unification Space actually check word order constraints because we never used input strings which contained more than one pair of brackets of the same type (e.g. '{}{}') and/or more than one type of embedding (e.g. '[<>]{}'). Thus word order constraints are in effect encoded in the bracket type feature.

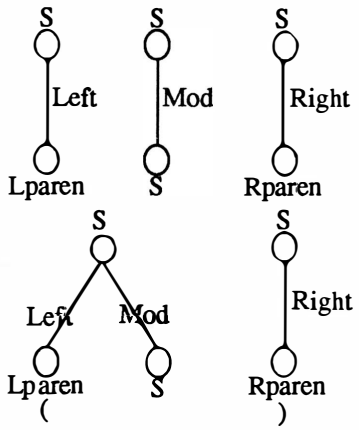


Figure 5. Segments of the grammar, and the lexical entries for '(' and ')'.

The actual simulations were run with 5 (levels) times 3 (dependency types) equals 15 different input strings. Each string was fed into the Unification Space 400 times. *The parameter settings were exactly equal to those used in the earlier Kempen & Vosse (1989) paper* [3]. No attempts have been made to find a set of parameter values yielding a better fit with Bach *et al.*'s empirical data.

The simulation results for the 15 sentences are displayed in Figure 7. They show the same general pattern as the comprehensibility ratings displayed in Figure 2 above. That is, (1) comprehensibility decreases with increasing depth of embedding, (2) center-embedded dependencies are harder than cross-serial dependencies, and (3) right-branching dependencies take a strong lead, being much easier to understand than both other constructions.

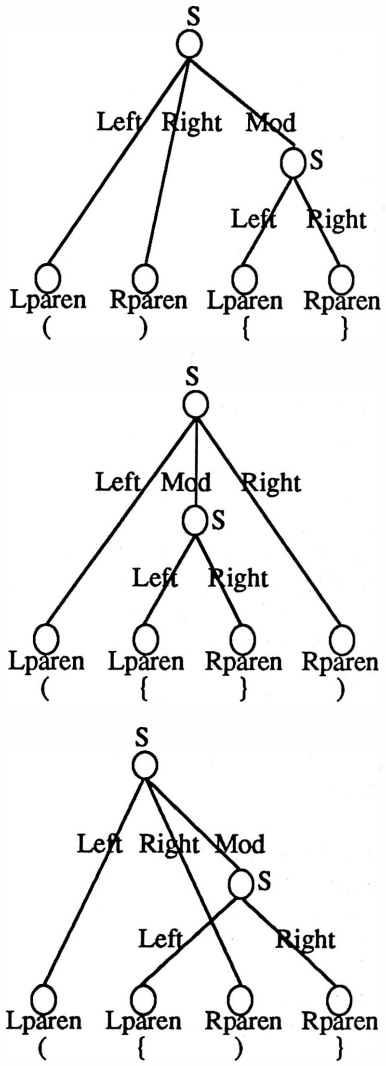


Figure 6. Example parse trees of level 2: respectively right-branching, center-embedded and cross-serial.

There are also differences between the human data and computer simulation, however. First of all, the comprehension scores for the three dependency types fan out more rapidly in our simulation than in the human subjects. Second, in the human data the first signs of a differentiation between sentence types manifest themselves already at level 2, whereas in our simulation the percentages start diverging at level 3 only. From our previous study we know that the Unification Space is rather sensitive to sentence length. If this applies to human readers as well, we could argue that our level 1 and level 2 scores are too

[3] For Chaos parameter *C* (not discussed in the present paper) we had four different values: .1, .2, .3 and .4. There were 100 runs for each value of *C*. In Figure 7 we show percentages averaged over *C* values.

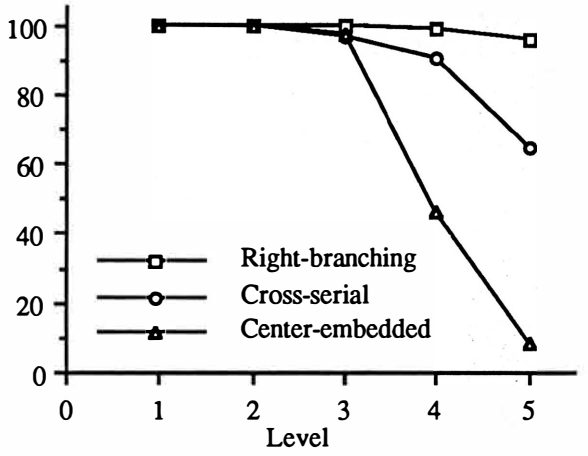


Figure 7. Percentages of correctly parsed strings for three types of dependency and five levels of depth.

good (in Bach *et al.*'s study, these levels were tested through sentence of 6 to 8 words long).

DISCUSSION

The simulation revealed a satisfactory fit between the empirical pattern of comprehensibility ratings observed by Bach *et al.* and parsability by the Unification Space. Since the model applied exactly the same grammar when processing the three types of dependencies, it follows that the empirical pattern can be explained in terms of the *different spatial-temporal arrangements* between the members of a dependency pair. No additional assumptions about differences between the syntactic structure underlying the three types of dependencies are needed.

To what extent are alternative computational models of human sentence processing capable of accounting for the empirical pattern? So far, Joshi's (1990) proposal is the only one reported in the literature. However, it is not clear how well this model behaves with respect to other psycholinguistic sentence processing phenomena such as Right Association, Minimal Attachment, Verb Frame Preferences and the like. Two other recent models (Gibson, 1990a,b,c; McRoy & Hirst, 1990) do address the latter phenomena but they pay no attention to cross-serial dependencies. So, as far as we know, there is no competing model of comparable wide coverage.

REFERENCES

Bach, Emmon, Colin Brown & William Marslen-Wilson (1986). Crossed and nested dependencies in German and Dutch: a psycholinguistic study. *Language and Cognitive Processes, 1*, 249-262.

De Smedt, Koen (1990). *Incremental sentence generation: Representational and computational aspects.* Ph.D. thesis, University of Nijmegen.

Frazier, Lyn (1987). Theories of sentence processing. In: Jay L. Garfield (Ed.), *Modularity in knowledge representation and natural language understanding.* Cambridge, MA: M.I.T. Press.

Gibson, Edward (1990a). Memory capacity and sentence processing. In: *Proceedings of the 28th Annual Meeting of the ACL*, Pittsburgh.

Gibson, Edward (1990b). Recency preferences and garden-path effects. In: *Proceedings of the 12th Annual Conference of the Cognitive Science Society*, Cambridge MA.

Gibson, Edward (1990c). A computational theory of processing overload and garden-path effects. *Proceedings of the 13th International Conference on Computational Linguistics* (COLING-90), Helsinki.

Joshi, Aravind K. (1990). Processing crossed and nested dependencies: an automaton perspective on the psycholinguistic results.. *Language and Cognitive Processes, 5(1)*, 249-262.

Kempen, Gerard (1987). A framework for incremental syntactic tree formation. In: *Proceedings of the 10th International Joint Conference on Artificial Intelligence* (IJCAI-87), Milan, 655-660.

Kempen, Gerard & Theo Vosse (1990). Incremental syntactic tree formation in human sentence processing: an interactive architecture based on activation decay and simulated annealing. *Connection Science, 1*, 273-290.

McRoy, Susan & Graeme Hirst. (1990). Race-based parsing and syntactic disambiguation. *Cognitive Science, 14*, 313-353.

Rayner, Keith & Alexander Pollatsek (1989). *The psychology of reading.* Englewood Cliffs, NJ: Prentice-Hall.

Sampson, Geoffrey (1986). A stochastic approach to parsing. In: *Proceedings of the 11th International Conference on Computational Linguistics* (COLING-86), Bonn.

Using inheritance in Object-Oriented Programming to combine syntactic rules and lexical idiosyncrasies

Benoît HABERT
Ecole Normale Supérieure de Fontenay Saint Cloud
31 avenue Lombart
F-92260 FONTENAY-AUX-ROSES FRANCE
internet: bh@litp.ibp.fr
bitnet: bh@frunip61.bitnet
Phone: (33) 1-47-02-60-50 Ext 415
Fax: (33) 1-47-02-34-32

ABSTRACT

In parsing idioms and frozen expressions in French, one needs to combine general syntactic rules and idiosyncratic constraints. The inheritance structure provided by Object-Oriented Programming languages, and more specifically the combination of methods present in CLOS, Common Lisp Object System, appears as an elegant and efficient approach to deal with such a complex interaction.

In parsing idioms and frozen expressions in French, one needs to combine general syntactic rules and idiosyncratic constraints. As a matter of fact, representing such an interaction via an inheritance lattice appears as an elegant and efficient approach. For the sake of explanation, English idioms will be used as examples. However this combination of syntactic rules and idiosyncratic behaviour via manipulations of the inheritance structure and the methods attached to it, has been designed for French compound adverbials. More than 6,000 compound adverbials have been listed and studied at LADL 1 (Gross, 1990). A lexicon-grammar of 2,525 compound adverbials coming from the LADL files has been used in parsing a test corpus of 72,000 words.

IDIOMS: A PECULIAR COMBINATION OF REGULARITIES AND IDIOSYNCRASIES

The semantics of idioms will not be accounted for here, since it is a controversial problem. LFG, GPSG and TAGs made quite different claims on this topic [2]. Within a syntactic category, it has been shown for French, at LADL, that frozen expressions are generally more numerous than 'free' ones: 20, 000 frozen verbs (12,000 free), 6,000 adverbials (1,500 free). More than 25, 000 compound nouns have been studied so far, but their number is far greater, as they constitute the major part of new terms in sublanguages (Grishman & Kittredge, 1986). A small proportion of frozen expressions have constituents existing only in such contexts (such as "umbrage" in "to take umbrage at NP"), or are taken from foreign languages ("a priori"), or follow forgotten rules. Apart from these marginal cases, idioms consist of the same words as the free phrases, and they follow the same syntactic rules: "in contrast", "by the way", for instance, are just ordinary PP. Furthermore, as shown for English by Wasow et al., 1982, and for French by Gross, 1988, 1990, the syntactic behaviour of idioms is much more systematic than is usually thought: 'transformations' apply to them. Some kind of 'meta-rules' must be then used to account for these related structures.

While following to a large extent the general syntactic rules, frozen expressions present idiosyncrasies. At a syntactic level, an idiom can accept a modifier ("in (loving)

1 Laboratoire d'Automatique Documentaire et Linguistique: Université Paris 7 and CNRS.

2 For Bresnan, 1982b, constituants of an idiom very often have a regular syntactic behaviour without contributing at all to the meaning of the whole expression. According to Gazdar et al.,, 1985, p 236-242, the semantic behaviour of idioms is more often compositional than has generally been assumed,. The approach of (Abeillé & Schabes, 1989) characterizes idioms by the combination of syntactic regularity and semantic non compositionality.

memory of"), or not (#"by the new way") [3]. It can require certain syntactic features for some of its constituents. For instance, it may need a certain type of determiner: "for the sake of" versus "#for a sake of". Lastly, an idiom is associated with fixed lexical items. Usually it is not possible to replace them by synonyms: #"by the road" versus "by the way". Since most of frozen expressions follow general syntactic rules, and since 'transformations' apply to them, it is not reasonable to try and process them in a first lexical step. Recognizing idioms belongs therefore to the whole syntactic analysis. Nevertheless their idiosyncratic features must be taken into account in rules.

STATING THE GENERAL BEHAVIOUR OF A FAMILY OF IDIOMS

OLMES [4] is a general parser written in CLOS [5] (Keene, 1989; Steele, 1990, p 770-864), and tested with the Victoria Day implementation of PCL [6] (provided by Xerox Laboratories), using Lucid Common Lisp 3.0.1, on a Sun 3 workstation, at LITP [7]. OLMES belongs to the active chart parser family. The input text can be parsed from left to right, or the other way round, or even both ways at the same time (around pivots). Top-down, bottom-up or bottom-up then top-down strategies are available. The rules used by OLMES follow the formalism created for PATR-II (Shieber, 1986), because it is a kind of "lingua franca" for unification-based grammars. Additional constraints can be associated with ordinary context-free rules so as to analyse mildly context-sensitive languages (Gazdar, 1988). Each symbol in the rule is the root of a Directed Acyclic Graph (DAG). In such category structures, each edge is labelled, and leads either to an atom or to another complex category structure (Gazdar et al., 1988).

For instance, a lot of adverbials in English use the following rule, in PATR-II form:

```
LHS -> RHS1 RHS2 RHS3
<LHS cat> = adv
<RHS1 cat> = prep
<RHS2 cat> = det
<RHS3 cat> = noun
<RHS2 agreement> = <RHS3 agreement>
```

The sequence of a first right-hand side symbol dominating a DAG with an edge "cat(egory)" having "prep(osition)" as its value, a second symbol with "cat" "det(erminer)", and a third symbol with "noun" as "cat" makes an "adv(erbial)". Additionally the second and the third symbol must share the same value for the feature "agreement".

A graphical equivalent could be:

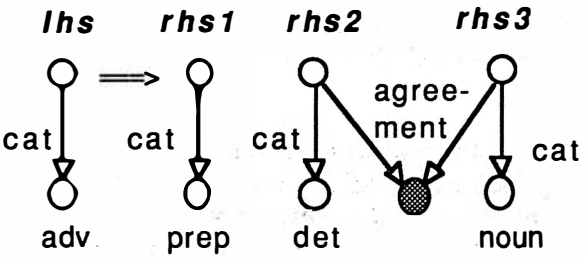


Figure 1: sequence of DAGs defining an adverbial

In the lexicon, one can find entries [8] such as:

```
a
   cat det
   cat-precisions
      determiner-type article
      article-type indefinite
at
   cat prep
by
   cat prep
in
   cat prep
end
   cat noun
   agreement
      number singular
```

---

[3] We use the same convention as (Gazdar et al., 1985): '#' indicates that a structure is acceptable, but with a literal meaning.

[4] Objects, Language, Means for Exploring and Structuring (Texts).

[5] Common Lisp Object System.

[6] Portable Common Loops

[7] Laboratoire d'Informatique Théorique et de Programmation: Université Paris 6, Université Paris 7 and CNRS.

[8] The features not relevant for the rule are not mentioned.

```
moment
   cat noun
   agreement
      number singular
my
   cat det
   cat-precisions
   determiner-type possessive
the
   cat det
   cat-precisions
      determiner-type article
      article-type definite
this
   cat det
   cat precisions
      determiner-type demonstrative
those
   cat det
   cat-precisions
      determiner-type demonstrative
   agreement
      number plural
way
   cat noun
   agreement
      number singular
```

The rule above would recognize as idioms "at the moment", "in a way", "in the end", using this toy lexicon. Note that the completed rule is more restrictive than the context-free part of it. The latter would accept "*by those way", the former would not, because "those" and "way" do not agree.

THE GRAMMAR: A NETWORK OF ACTIVE AGENTS ENCAPSULATING CONSTRAINTS

The context-free rules of the grammar are represented by a network of classes. Each class in the network corresponds to an occurrence of a symbol, whether terminal or not, appearing in the grammar. The topology of the network mirrors exactly the strategy (top-down versus bottom-up) and the direction of exploration (left-right, right-left or bi-directional) chosen by the user when compiling the grammar. This approach extends the work done within the actor paradigm by Yonesawa & Ohsawa, 1990.

There are two main classes: active and inactive. An inactive agent corresponds to a (possibly partial) constituent which has been found. For instance, for each left-hand side symbol in the grammar, a class is created inheriting from the inactive agent class. The active agents correspond to the right-hand side symbols of the grammar. Each of them is searching for a constituent meeting certain constraints, as defined in the corresponding DAG in the rule. If it finds such a constituent, it then creates an instance of the class corresponding to the following symbol in the right-hand side part of the rule. When the last active agent of the rule "succeeds", it creates an instance of the class corresponding to the left-hand side of the rule. The pivot of the rule is the symbol starting the whole analysis. It need not be the left-most one.

For the rule above, in bottom-up parsing, four classes are defined: `LHS-1`, `RHS1-2`, `RHS2-3`, `RHS3-4`, respectively (figure 2). `RHS1-2`, `RHS2-3` and `RHS3-4` are subclasses of `LHS-1`, their instances will be active agents examining the text from right to left. The pivot of the rule is the class `RHS3-4` (in bold font), corresponding to a noun.

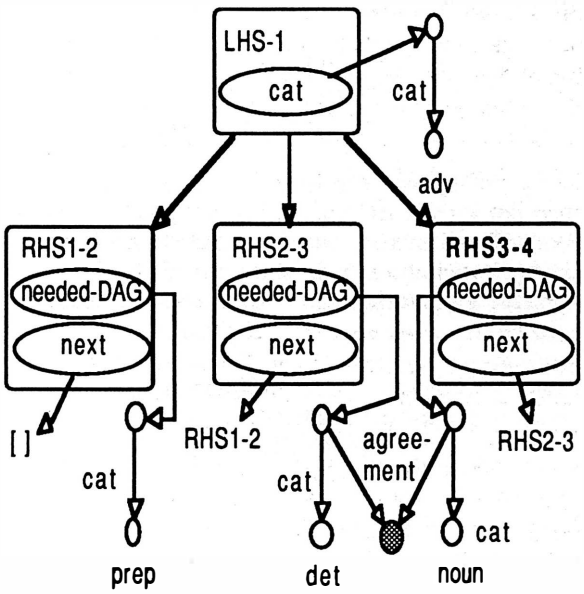


Figure 2: classes resulting from the compilation

To indicate that a word can belong to the type of idiom described in the rule, the lexicon associates the class-name `RHS3-4` with this word. It could be the case for the word "moment". In a bottom-up analysis, for each occurrence of "moment" in the input text,

OLMES creates an instance of `RHS3-4`. This instance searches for a noun, and finds it: "moment". It creates an instance of `RHS2-3` which examines the word on the left of "moment", and which stores a partial parse tree. If this word is a determiner, and has a feature "agreement" matching with the corresponding feature of "moment", the new partial parse tree is transmitted to the instance of `RHS1-2` which is then created and whose constraints are matched against the word on the left of the determiner found by the instance of `RHS2-3`. In the case that the instance of `RHS1-2` finds a preposition, it then creates an instance of `LHS-1` storing the complete parse tree and the additional information gathered from the unification on the rest of the DAGs.

Changing the grammar rules from sequences of 'passive' labels to a network of active classes makes it possible to increase as necessary the knowledge the instances of these classes can utilise, and to use inheritance not only in the lexicon (Shieber, 1986), but in the grammar rules as well.

USING THE INHERITANCE STRUCTURE TO TAKE IDIOSYNCRASIES INTO ACCOUNT

The rule stated above is not restrictive enough. For instance, it would parse as an idiom "by a way" in the sentence: "he arrived by a way new to me". It would be rather an unsatisfactory approach to create as many rules as combinations found between the preposition and the type of determiner used in such idioms. What we need instead is a means to adjoin new constraints to the set of conditions defined in the rule, in a modular way, that is, using inheritance. In the CLOS philosophy, it means that some 'mixin' classes are created. Such classes are not intended to have instances on their own. On the contrary, they are only used as constituents (super-classes) in defining more specialized classes.

For instance, one can define the following 'mixin' classes (see figure 3). Each 'mixin' class used to specialize the rule has a method `constraints` which states particular constraints on the determiner. The content of this method (in PATR form) follows the class name, below.

```
det-article
   <RHS3 det1 cat-precisions
determiner-type> = article

det-definite-article (subclass of
det-article)
   <RHS3 det1 cat-precisions
article-type> = definite

det-indefinite-article (subclass of
det-article)
   <RHS3 det1 cat-precisions
article-type> = indefinite

det-possessive
   <RHS3 det1 cat-precisions
determiner-type> = possessive

det-demonstrative
   <RHS3 det1 cat-precisions
determiner-type> = demonstrative
```

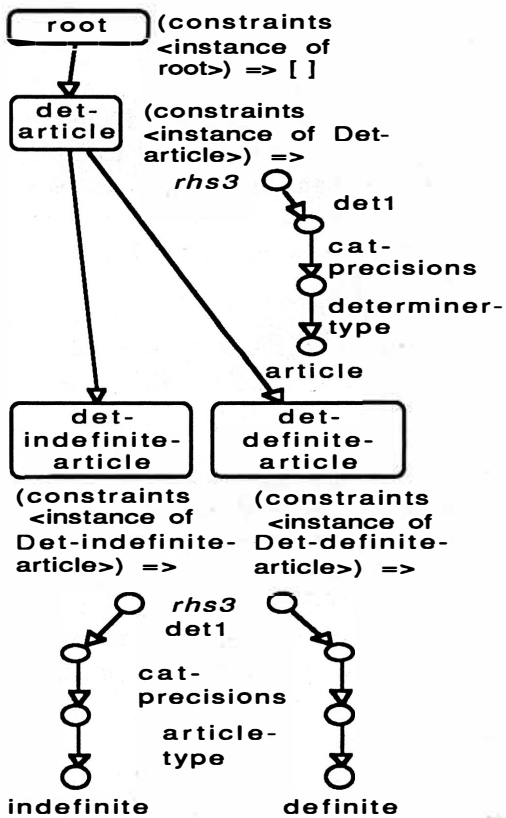


Figure 3: Some classes for the constraints on determiners

The rule given above (figure 1) is slightly redefined : from now on, the pivot transmits to

the RHS1 the form of preposition, and to the RHS2 precisions on the type of determiner which is needed (the dark nodes indicate this sharing of values in figure 4).

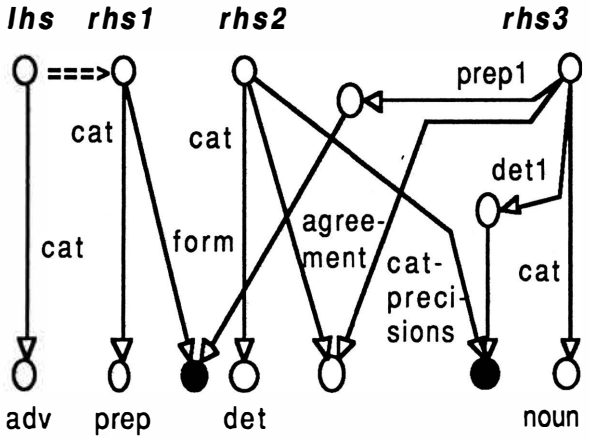


Figure 4: Redefined rule for adverbials

It is now possible to create final classes for the pivots of the idioms:

- `adv=prep_det-definite-article_noun`, subclass of `RHS3-4` and `det-definite-article`. E.g.: by the way.

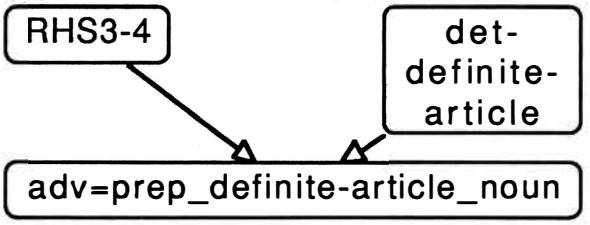


Figure 5: An example of final class

- `adv=prep_det-indefinite-article_noun`, subclass of `RHS3-4` and `det-indefinite-article`. E.g.: in a way.
- `adv=prep_det-demonstrative-noun`, subclass of `RHS3-4` and `det-demonstrative`. E.g.: in this respect.
- `adv=prep_det-possessive_noun`, subclass of `RHS3-4` and `det-possessive`. E.g.: in my opinion.

Of course, it could have been possible to define mixin classes to deal with constraints on the preposition. Such classes would have looked like:

`prep-in`

```
   <RHS1 prep1 form> = in
prep-by
   <RHS1 prep1 form> = by
```

and so on.

It should be noted that the constraints on the preposition and the conditions on the determiner are not on the same level. The latter are in a way more syntactic: some syntactic properties of the idiom as a whole depend on the nature of the determiner. The form of the initial preposition is purely idiosyncratic. It does not even always contribute to the meaning of the expression. For that reason, the way to specify the initial preposition does not use inheritance. A list of associations *{<parameter><value>}* is being used at the initialization of the instance of a given pivot class to deal with such litteral constraints. For instance, the list "RHS1 by" will trigger the adjunction of:

```
   <RHS1 prep1 form> = by
```

to the conditions specified for the idiomatic rule.

Usually, in an Object-Oriented programming language, when a method is called for an instance of a class, and there are different methods of the same name linked with the ancestors of this class, the most specific method is actually used, overriding the other ones. For instance, calling `(constraints det-definite-article-1)`, `det-definite-article-1` being an instance of `det-definite-article`, would yield 9:

```
   <RHS3 det1 cat-precisions
article-type> = definite
```

As shown in figure 6, three methods are applicable (inside the grey frame): `constraints` of `Det-definite-article` (`det-definite-article-1` is an instance of this class), `constraints` of `Det-article` (an instance of `Det-definite-article` is a `Det-article`) and `constraints` of `root` (for the same reason). The last one (in bold font) shadows the others.

9 We do not use the actual Lisp syntax for the result, as it is not relevant.

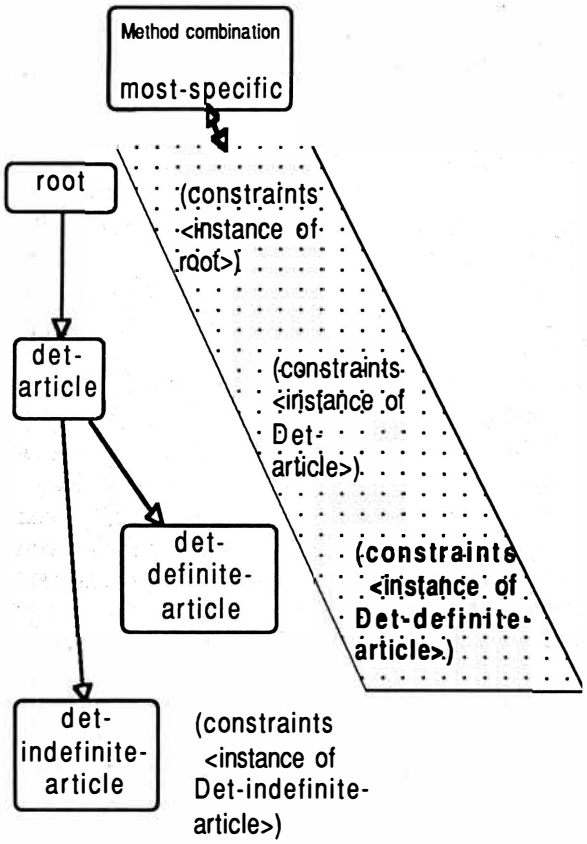


Figure 6: Standard method combination

One of the salient characteristics of CLOS, inherited from its ancestors, COMMON LOOPS (Bobrow et al., 1986) and NEW FLAVORS, is the control given over the combination of methods having the same name and present in the super-classes of a given class (Keene, 1989) [10] . In this case, it is possible to specify that all the methods `constraints` accessible from a given class should be called in turn, and the final result should be the addition of all the returned values. With this combination of methods, the result of the function call `(constraints det-definite-article-1)` would be (figure 7):

```
   <RHS3 det1 cat-precisions
determiner-type> = article
```

[10] When the most specific method represents nothing but an addition to the action of one super-method, it is generally possible in an Object-Oriented Programming Language to combine it with this super-method, so as to share common behaviours.

```
   <RHS3 det1 cat-precisions
article-type> = definite
```

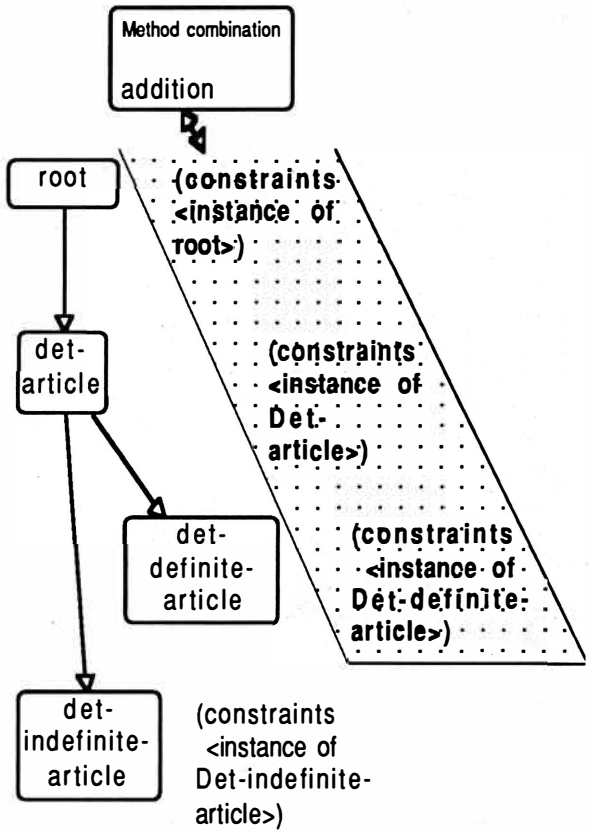


Figure 7: Special method combination

This method combination provides a means to add constraints present in the inheritance lattice, and only the relevant ones.
In the lexicon, the entries for pivots of adverbials could mention (among other information):

```
moment
   adv=prep_definite-article_noun
RHS1 at

opinion
   adv=prep_possessive_noun RHS1 in

way
   adv=prep_definite-article_noun
RSH1 by
   adv=prep_indefinite-article_noun
RHS1 in
```

When coming across "way" in the input text, OLMES would therefore create one instance of each class. For example, in the case of the instance of `adv=prep_definite-`

article_noun, because this class is a subclass of det-definite-article, and because the method combination for constraints is redefined, the constraints inherited via det-definite-article and det-article are added to the general constraints defined in the rule and inherited through RHS3-4. The arguments following the name of the class are used as well. In the end, the parser will actually try the following rule (figure 8):

```
   LHS -> RHS1 RHS2 RHS3
   <LHS cat> = adv
   <RHS1 cat> = prep
   <RHS1 form> = <RHS3 prep1 form>
   <RHS2 cat> = det
   <RHS3 cat> = noun
   <RHS3 prep1 form> = by
   <RHS2 agreement> = <RHS3
agreement>
   <RHS2 cat-precisions> =
<RHS3 det1 cat-precisions>
   <RHS3 det1 cat-precisions
determiner-type> = article
   <RHS3 det1 cat-precisions
article-type> = definite
```

(The basic constraints are in normal font, the inherited ones in bold font, and the parametrized ones are underlined.)

This rule will accept "by the way", but will reject "by a way", "in the way" ... The rule and the parameters for "moment" would allow the parsing of "at the moment", and those for "opinion" the acceptance of "in my opinion"...

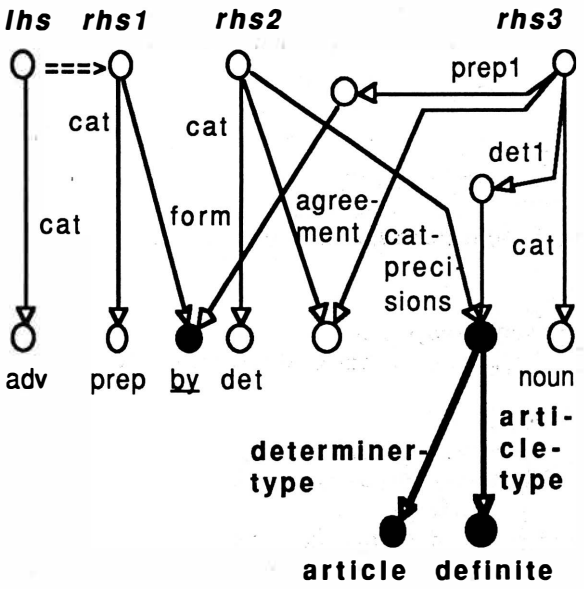


Figure 8: Sequence of DAGs for the final rule

This very simple example does not do justice to the complexity of syntactic and lexical properties of adverbial idioms. However it stresses the hierarchy of these features, and the way in which the inheritance graph can, at the same time, mirror this structure and take advantage of it. Note that the 'mixin' classes defined above are useful as such. They constitute the primitives to complete the basic syntactic rules. They correspond to organized constraints which are interesting on their own, as they can be reused in different contexts. For instance, another rule for adverbials is:

```
   LHS -> RHS1 RHS2 RHS3 RHS4
   <LHS cat> = adv
   <RHS1 cat> = prep
   <RHS1 form> = <RHS3 prep1 form>
   <RHS2 cat> = det
   <RHS3 cat> = noun
   <RHS2 agreement> = <RHS3
agreement>
   <RHS2 cat-precisions> = <RHS2
det1 cat-precisions>
   <RHS4 cat> = prep
   <RHS4 form> = <RHS3 prep2 form>
```

A class would be created for each symbol of the rule. For example, the class corresponding to the pivot (the noun) is RHS4-9. New specialized classes are then defined:

- adv=prep_definite-article_noun_prep (super-classes: RHS4-9, det-definite-article)
- adv=prep_indefinite-article_noun_prep (super-classes: RHS4-9, det-indefinite-article)...

And the lexicon now has entries like:

eye
adv=prep_indefinite-article_noun_prep RHS1 with RHS4 to

form
adv=prep_definite-article_noun_prep RHS1 in RHS4 of

which could recognize "with an eye to" and "in the form of", respectively. It is possible in OLMES to express that a certain word can enter different linked syntactic structures at the same time, thus providing 'meta-rules'. One of theses families of rules is the class: [adv = prep definite-article noun prep + adv = prep possessive-determiner noun] gathering the following rules

adv=prep_definite-article_noun_prep
adv=prep_det-possessive_noun

And, in the lexicon, the entry time mentions:

[adv = prep definite-article noun prep + adv = prep possessive-determiner noun] RHS1 for RHS4 of

Therefore, the parser, when finding time in the input text, creates an instance of the class [adv = prep definite-article noun prep + adv = prep possessive-determiner noun] , which in turn creates an instance of adv=prep_definite-article_noun_prep and an instance of adv=prep_det-possessive_noun. This instance of [adv = prep definite-article noun prep + adv = prep possessive-determiner noun] also transmits to them the correct values for the parameters RHS1 and RHS4, possibly leading to the parsing of for the sake of or for its sake.

RELATED WORK: PARSING IDIOMS IN TREE ADJOINING GRAMMARS (TAGS)

Recent work, within the TAG formalism (Abeillé, 1990; Abeillé & Schabes, 1989), claimed that idioms should be parsed during the whole syntactic analysis, using the same formal devices as for parsing non idiomatic expressions. This approach uses a slightly modified version of TAGs, namely lexicalized TAGs, in which each 'rule', i.e. each tree, is anchored with a lexical item. In fact, there are no separate phrase structure rules any more: they are collapsed into the lexicon. Only the relevant rules are used while parsing, as they are triggered by lexical items. Meta-rules are provided by means of families of trees. The trees corresponding to idioms include several lexical items: take and bucket in the case of to take the bucket . As an additional filtering, the search of an idiom is triggered only if all the lexical heads are actually present in the sentence, in the right order.

As a matter of fact, giving the rules a pivot and associating the words in the lexicon with these pivots, as shown above, is a first step in lexicalizing a grammar. On the other hand, Lexicalized TAGs do not use phrase structure rules any more, but trees directly stating to any depth the constituents needed, their structure, and possibly their lexical heads. For this very reason, the TAG formalism deals with idioms in a more natural and powerful way. For the sake of explanation, the rules given in this paper are flattening the structure of the phrases. In order to give to the relevant idioms the same structure as the corresponding free phrases, one would need some complex transmission of features among related rules (Habert, 1991).

STRUCTURING GRAMMARS VIA INHERITANCE

Relying to such an extent on the inheritance structure partly breaks the decentralization rule which is central to object oriented programming 11. When slightly modifying a class, here is a risk of triggering a

11 (Meyer, 1988, p 251) In most cases, clients of a class should not need to know the inheritance structure that led to its implementation.

chain reaction of changes. As Sakkinen, 1989, states:

> Features aiming at "exploratory programming" need not necessarily make the programmer into a Vasco de Gama or an Amundsen; (s)he may well become Alice in Wonderland, never knowing what metamorphoses some seemingly innocent act may cause.

The danger is a real one. Nevertheless, so far, it has been most beneficial to take advantage of the inheritance structure to portray the linguistic knowledge we are dealing with. In doing so, we stress the classification tools present in Objet-Oriented Programming Languages: the inheritance lattice is used to progressively constrain the class of the solution (Wegner, 1987). This approach uses a unification-based formalism with a clear-cut distinction between phrase structure rules and subcategorization frames. In spite of this, it combines properly the generalizations stated by the syntactic rules and additionnal constraints necessary to account for the idiosyncrasies that the idioms show. This solution is by no means limited to frozen expressions. It contributes to a clear expression of the complex interactions found in the grammar between syntactic and lexical rules (Abeillé 90). It is thus worth investigating the ways in which inheritance can help in structuring not only the lexicon but also the grammar.

AKNOWLEDGMENTS

I greatly benefited from discussions with F.-X. Testard-Vaillant (LITP), Pierre Fiala (ENS de Fontenay Saint Cloud), and Anne Abeillé (LADL) on Object-Oriented Programming, Idioms and TAGs respectively.

REFERENCES

Abeillé Anne
1990 "Lexical and syntactic rules in a tree adjoining grammar", ACL'90

Abeillé Anne, Yves Schabes
1989 "Parsing Idioms in Lexicalized TAGs", EACL'89.

Bresnan Joan
1982a (editor) The mental representation of grammatical representation, The MIT Press.
1982b "The passive in Lexical Theory", (Bresnan, 1982a, p 2-86).

Bobrow Daniel G., Kenneth Kahn, Gregor Kickzales, Larry Masinter, Mark Stefik and Frank Zdybel
1986 "CommonLoops: merging Lisp and Object-Oriented Programming, OOPSLA'86.

Gazdar Gerald
1988 "Applicability of Indexed Grammars to natural languages", in Natural language parsing and linguistic theories, Reyle and Rohrer editors, D. Reidel Publishing Company.

Gazdar Gerald, Ewan Klein, Geoffrey Pullum, Ivan Sag
1985 Generalized Phrase Structure Grammar, Harvard University Press.

Gazdar Gerald, Mellish Chris
1989 Natural Language Processing in Lisp, Addison-Wesley.

Gazdar Gerald, Geoffrey K. Pullum, Robert Carpenter, Ewan Klein, Thomas E. Hukari, Robert D. Levine
1988 "Category structures", Computational Linguistics, Vol. 14, #1.

Grishman Ralph, Richard Kittredge (editors)
1986 Analyzing language in restricted domains: sublanguage description and processing, Lawrence Erlbaum Associates.

Gross Maurice
1988 "Sur les phrases figées complexes du français", Langue Française 77.
1990 Grammaire transformationnelle du français: 3 - syntaxe de l'adverbe, ASTRIL.

Habert Benoît
1990 "Controlling the generic dispatch to represent domain knowledge", Proceedings of the third CLOS users and implementors workshop, OOPSLA'90.
1991 Langages à objets et analyse linguistique, Doctoral thesis.

Keene Sonya E.
1989 Object-Oriented Programming in Common Lisp, Addison Wesley.

Kay Martin
1985 "Parsing in Functional Unification Grammar", in Natural language parsing, D. Dowty, L. Karttunen and A. Zwicky editors, Cambridge University Press.

Kickzales Gregor, Luis Rodriguez
1990 "Efficient method dispatch in PCL", Proceedings of the 1990 conference on Lisp and Functional Programming.

Meyer Bertrand
1988 Object-Oriented Software Construction, Prentice Hall.

Pollard Carl, Ivan A. Sag
1987 Information-based syntax and semantics, Vol 1: Fundamentals, CSLI.

Sakkinen Markku
1989 "Disciplined Inheritance", ECOOP'89

Shieber Stuart
1986 An introduction to unification-based approaches to grammar, CSLI.

Steele Guy L.
1990 Common Lisp: The Language, 2nd edition, Digital Press.

Wegner Peter
1987 "The Object-Oriented Classification Paradigm", in Research Directions in Object-Oriented Programming, The MIT Press.

Wasow Thomas, Ivan A. Sag, Geoffrey Nunberg
1982 "Idioms: an interim report", Proceedings of the 13th International Congress of Linguists,

Yonesawa Akinori, Ichiro Ohsawa
1990 "Object-Oriented Parallel Parsing for Context-Free Grammars", in ABCL: an Object-Oriented Concurrent System, Akinori Yonezawa editor, The MIT Press.

February 14, 1991

Session A

AN LR(k) ERROR DIAGNOSIS AND RECOVERY METHOD

Philippe Charles
IBM T.J. Watson Research Center
P.O. box 704,
Yorktown Heights, N.Y 10598

Abstract

In this paper, a new practical, efficient and language-independent syntactic error recovery method for LR(k) parsers is presented. This method is similar to and builds upon the three-level approach of Burke-Fisher [11]. However, it is more time- and space-efficient and fully automatic.

1 Introduction and Overview

1.1 The Parsing Framework

An LR *parsing configuration* has two components: a state stack and the remaining input tokens. This method assumes a framework in which the parser maintains a state stack, denoted *stack*, and a fixed number of input symbols. These symbols include the current token or *lookahead*, denoted *curtok*, the token immediately preceding the current token, denoted *prevtok*, and an input buffer, denoted *buffer*, containing a predetermined number of the input tokens following *curtok*. A number of attributes are associated with each input symbol such as its class, its location within the input source, its character string representation, etc ...An input symbol together with all its attributes is referred to as a *token element*. Each state q in the state stack is also associated with certain attributes including the grammar symbol that caused the transition into q (called the *in_symbol* of q), and the location of the first input token on which an action was executed on q.

An LR parsing configuration may be represented by a string of the form:

$$q_1, q_2, \ldots, q_m \quad | \quad t_1, t_2, \ldots, t_n.$$

The sequence to the left of the vertical bar is the content of the state stack, with q_m at the top; $q_1 \ldots q_m$ is a valid sequence of states in the LR parsing machine. The sequence to the right of the vertical bar is the unexpended input. Each element t_i represents the class of a corresponding input symbol. The symbol t_1 represents the class of the current token, t_2 represents the class of the successor of *curtok*, etc. The symbol t_0 which is not shown above represents the class of *prevtok*.

For simplicity, it will be assumed that the grammar used to construct the parser is LR(1), but this method is applicable to all forms of LR(k) parsers.

1.2 Error Recovery

A parsing configuration in which no legal action is possible is called an *error configuration*. When an error configuration is reached, the error recovery procedure is invoked. Its role is to adjust the configuration so as to allow the parser to advance a minimum predetermined distance in the input stream, usually two or three tokens past the repair point. The token on which the error is detected is referred to as the *error token* and the state in which the error is detected is called the *error state*.

Three kinds of recovery strategies are used. They are:

- *Primary recovery.* A single symbol modification of the source text; i.e., the insertion of a single symbol into the input stream, the deletion of an input token, the substitution of a grammar symbol for an input token or the merging of two adjacent tokens to form a single one. Previous authors [7][11] have used a more restricted form of primary recovery involving only terminal symbols as repair candidates.
- *Secondary recovery.* Deletion of as small a sequence of tokens as possible in the vicinity of the error token or replacement of such a sequence with a nonterminal symbol. This approach can be viewed as an **automatic** generalization of the *error productions* method described in [3].
- *Scope recovery.* A scope is a syntactically nested structure such as a parenthesized expression, a block or a procedure. In scope recovery, the strategy is to recover by inserting relevant symbols into the text to complete the construction of scopes that are incompletely specified.

```
 1. program TEST(INPUT, OUTPUT);
 2.     var X,Y: array[] of integer;
                  ^
*Error: index_list expected after ...
 3. begn
    ^
*Error: misspelling of BEGIN
 4. 1:   x := y,
                ^
*Error: ; expected instead of this token
 5.      if x == b then begin
                ^
*Error: Unexpected symbol ignored
 6.         go to 1;
            <--->
*Error: Symbols merged to form GOTO
 7.      a := ((b + c)
                   ^
                   ^
*Error: ")" inserted to complete phrase
*Error: "END" inserted to complete ...
 8. end.
```

Figure 1: Primary phase recoveries

```
 1. program P(INPUT,OUTPUT);
 2.     procedure P(X:INTEGER):integer;
                               <------>
*Error: Unexpected input discarded
 3.     begin
 4.     end;
 5. begin
 6.    if count[listdata[sub] := 0 then
                              ^
                              ^
*Error: "]" inserted to complete phrase
*Error: invalid relational_operator
 7.        a := ((b + c ]];
                   ^
                   ^
                     <>
*Error: ")" inserted to complete phrase
*Error: ")" inserted to complete phrase
*Error: Unexpected input discarded
 8. end.
```

Figure 2: Secondary phase recoveries

This error recovery scheme consists of two phases called *Primary phase* and *Secondary phase*. In the *Primary phase*, an attempt is made to recover with minimal modification of the remaining input stream. Figure 1 shows some examples of primary phase recoveries. In the *Secondary phase*, more radical approaches involving removal of some left context (state stack) information as well as multiple deletion of tokens from the input stream (right context) are attempted. Figure 2 shows some examples of secondary phase recoveries.

1.3 Error Detection

A canonical LR(k) parser has the capability of detecting an error at the earliest possible point. However, because of their size, canonical LR(k) parsers are seldom used. Instead, variants such as LALR(k) and SLR(k) (usually $k = 1$), invented by DeRemer [1][2] are used. These LR variants, in part, solve the space problem by always using the underlying LR(0) automaton. However, certain states in these parsers usually contain reduce actions that may be illegal, depending on the actual context. Illegal reduce actions do not cause the resulting parser to accept illegal inputs, but they prevent it from always detecting errors at the earlier possible point. This problem is usually compounded by a space-saving technique known as *default reductions* which is often used in compressing parsing tables. To apply the default reductions technique, the most common rule by which the parser can reduce in each state is chosen as a default action for that state and all the reduce actions by that rule are removed from the parsing table. Another undesirable side effect of using default reductions is that it is no longer possible to compute, from the parsing table, the set of terminal symbols on which valid actions are defined in a given state. The inability to detect errors as soon as possible and to obtain a set of viable terminal candidates for a given state is very problematic for error recovery.

Furthermore, even with a canonical LR(k) parser, the ability to detect an error at the *earliest possible point* only guarantees that the prefix parsed up to that point is correct. Therefore, it is possible that the token on which an error is detected is not the one that is actually in error. Consider the following Pascal declaration:

```
FUNCTION F(X:TINY, Y:BIG, Z:REAL);
```

In this example, it is very difficult to deduce the actual intention of the programmer, but a simple substitution of the keyword "PROCEDURE" for

the keyword "**FUNCTION**" would solve the problem. However, the error is not detected until the semicolon (;) is encountered or 15 tokens later.

In [11], Burke and Fisher introduced a *deferred parsing* technique where two parsers are run concurrently: one that parses normally and another that is kept at a fixed distance (measured in terminal symbols) back. When an error is encountered, error recovery is attempted at all points between the two parsers. This approach avoids the premature reductions problem and solves, in part, the problem of late detection of errors. However, the overhead of the two parsers penalizes correct programs.

In this method, a new LR driver routine called *deferred driver* is introduced. This new driver can effectively detect an error at the earliest possible point even if the parser contains default reductions. It can also be adapted to defer parsing actions on a fixed number of tokens with very little slow-down on correct programs. To achieve this goal, an additional state stack is required for each deferred symbol. Thus, in practice, one must restrict the number of symbols on which actions are deferred.

The method also relies on having two mappings: *t_symbols* and *nt_symbols*, statically constructed, which yield for each state, a subset of the terminal and nonterminal symbols, respectively, on which an action is defined in the state in question. These subsets are the smallest subsets of viable error recovery *candidates* for each state. Their computation will be discussed later.

The remainder of this paper is organized as follows:

- detailed description of the new driver
- presentation of various recovery techniques
- discussion of how to apply these recovery techniques
- concluding remarks

2 The Driver

An important improvement that can be made to an LR(k) automaton is the removal of *LR(0) reduce states*. An LR(0) reduce state is a state that contains only reduce actions by a particular rule. If a representation of the parsing tables with default action is used, then the parser will never consult the lookahead symbol when it is in one of these states. Thus, such states may be completely removed from the parser by introducing a new parsing action: *read-reduce*. The read-reduce action comprises a read transition followed by a reduction. A read-reduce action is referred to as a *shift-reduce* when

```
# let #x denote the number of elements in a
# sequence x. rhs and lhs are maps that yield the
# size of the right-hand side and left-hand side
# symbol of a given rule, respectively. ACTION
# and GOTO are the terminal and nonterminal
# parsing functions, respectively.
1. function lookahead_action(stk, tok, pos);
2. {    pos := #stk.state;
3.      top := pos - 1;
4.      act := ACTION(stk.state[pos], tok);
5.      while act is a reduce action do
6.      {   do
7.          {   top := top - rhs[act] + 1;
8.              if top > pos then
9.                  s := tstk[top];
10.             else  s := stk.state[top];
11.             act := GOTO(s, lhs[act]);
12.         } while act is a goto-reduce action;
13.         tstk[top+1] := act;
14.         act := ACTION(act, tok);
15.         pos := min(pos, top);
16.     }
17.     return act;
18. }
```

Figure 3: *lookahead_action* function

the symbol X in question is a terminal symbol and as a *goto-reduce* action when X is a nonterminal.

The removal of LR(0) reduce states from an LR automaton does not cause premature reductions. Moreover, the execution of a read-reduce action is always followed by a sequence of zero or more goto-reduce actions, and finally, by a goto action. All of these actions may also be executed without deferral.

When the parser executes a reduce action in a non-LR(0) reduce state, that action is also followed by goto-reduce actions and a final goto action. If the reduce action in question is an illegal action, executed by default, then all the associated goto-reduce and goto actions following it are also illegal moves. To complicate matters, the goto action may be followed by a sequence of reduce actions on empty rules, each followed by its associated goto-reduces and goto action. In such a case, all actions induced by the lookahead symbol must be invalidated and the original configuration of the parser (prior to the initial reduction) must be restored.

One way to achieve this goal is as follows. When a reduce action is encountered, make a copy of the state stack into a temporary stack and simulate the parser using the temporary stack until either a shift, shift-reduce or error action is com-

```
stk.state := [start_state];
loop do
{   ppos := 0;   pstk := [ ];
    npos := 0;   nstk := [ ];
    stk.loc[#stk.state] := curtok.loc;
    tstk := stk;
    act := lookahead_action(tstk, t1, pos);
    while act ≠ error and act ≠ accept do
    {   nstk[npos+1..] := tstk[npos+1..];
        stk.loc[pos+1..] :=
            [curtok.loc : i in [pos+1..#nstk]];
        if act is a shift-reduce action then
        {   top := #nstk;
            do
            {   top := top - rhs[act] + 1;
                act := GOTO(nstk[top], lhs[act]);
            } while act is a goto-reduce action;
            nstk[top+1..] := [act];
            pos := min(pos, top);
        }
        act := lookahead_action(nstk, t2, npos);
        if act ≠ error then
        {   get next token;
            pstk[ppos+1..] := stk.state[ppos+1..];
            ppos := pos;
            stk.state[pos+1..] := nstk[pos+1..];
            pos := npos;
        }
    }
    if act =accept then
        return;
    error_recovery();
}
```

Figure 4: *Driver with 3 deferred tokens*

puted on the lookahead symbol. If the first non-reduce action computed on the lookahead is valid, the temporary state stack is copied into the state stack and the parsing can continue. Otherwise, the error recovery routine is invoked with the unadulterated state stack. This idea captures the essence of what needs to be done, but it is too costly for practical use.

Instead of copying the information, the temporary stack is used to hold the values of the contiguous elements of the state stack that have been added or rewritten. If the moves turn out to be valid, then only the added or rewritten elements are copied to the state stack. Otherwise, the original configuration is passed to the error recovery routine. This idea is illustrated in the *lookahead_action* function of Figure 3, written in pseudo-code.

The *lookahead_action* function always returns the first non-reduce action computed on the lookahead symbol. If that action is valid, the state sequence of the new configuration consists of the elements $1..pos$ of $stk.state$ and the elements $pos + 1..top + 1$ of $tstk$.

A parser with actions deferred on one token can be constructed as follows. Starting with the initial configuration, the parser advances through the input stream one token at a time after verifying that the token in question is a valid input by invoking the *lookahead_action* function. When the *lookahead_action* function is invoked with a valid lookahead it returns either a shift or a shift-reduce action which is processed immediately. As mentioned earlier, shift-reduce actions and all their associated goto-reduce and final goto actions may be processed without deferral. After successfully processing a token, the next token is read in and the process is repeated on the new configuration. If, on the other hand, the *lookahead_action* function returned the error action, the state stack is not updated and the error recovery routine is invoked instead.

A driver routine can be constructed, using the *lookahead_action* function, to defer parsing actions on n tokens given n state stacks. In experiments with this method, parsing has been deferred for three tokens. The three stacks that are used are: pstk which captures the configuration of the parser prior to processing any action induced by *prevtok*, stk which captures the configuration prior to processing actions induced by *curtok*, and nstk which captures the configuration prior to processing actions induced by the successor of *curtok*. Associated with each of these stacks are three integer variables: ppos, pos and npos which are used to mark the position of the top element in the corresponding stack that is still valid after the actions induced by the relevant lookahead symbol are applied. Figure 4 shows the body of a driver routine with actions deferred on three input symbols.

3 Recovery Strategies

Each recovery attempt is called a *trial*. The effectiveness of a recovery is evaluated using a validation function: *parse_check*, which indicates how many tokens in the input buffer can be successfully parsed after the repair in question is applied: *parse_check distance*. A recovery trial is not considered successful unless the parse_check distance is greater than or equal to a certain value, called *min_distance*. Experiments have shown that a good choice for *min_distance* is 2 [11].

The *parse_check* function is essentially an LR driver that simulates the parse until it has either shifted all the tokens in the buffer, completed the parse successfully, or reached a token in error.

In the following subsections, algorithms for optimizing the necessary error recovery information and implementing the three different recovery strategies are presented.

3.1 Primary Recovery

Given a configuration: $q_1, q_2, \ldots, q_m \mid t_1, t_2, \ldots, t_n$, where t_1 is assumed to be the error token, the primary recovery finds the best possible *primary repair* (if any) for that configuration. The selection of a best primary repair is based on three criteria:

- the parse_check distance
- the *misspelling index*
- the order in which the trials are performed.

The misspelling index is a real value between 0.0 and 1.0 that is associated with each primary recovery trial. When a new token is substituted for the error token - a *simple substitution*, a misspelling function is invoked to determine the misspelling index; i.e., the relative proximity of the two tokens in question expressed as a probabilistic value. For other kinds of recoveries, the misspelling index is set to a constant value depending on the recovery in question and other conditions. This will be discussed later.

Primary recoveries are attempted in the following order: merging of the error token (t_1) with its successor (t_2); deletion of t_1; insertion of each terminal candidate in $t\_symbols(q_m)$ before t_1; substitution of each legal terminal candidate in $t\_symbols(q_m)$ for t_1; insertion of each nonterminal candidate in $nt\_symbols(q_m)$ before t_1; and, finally, substitution of each nonterminal candidate $nt\_symbols(q_m)$ for t_1; For now, one can assume that for a state q, $t\_symbols(q)$ and $nt\_symbols(q)$ yield the sets of all terminal and nonterminal symbols, respectively, on which actions are defined in q. Optimization of these sets is discussed in section 3.3.

As the trials are performed, the primary recovery routine keeps track of the most succesful trial. Initially, the merge recovery is chosen since it is attempted first. If a subsequent recovery yields a larger parse_check distance than the previously chosen recovery or it yields the same parse_check distance but with a greater misspelling index, then it is chosen instead as the best recovery candidate.

For the merge trial, the character string representation of t_2, is concatenated to the character string representation of t_1 to obtain a merged string s. A test is then performed to determine if s is the character string representation of some $t \in t\_symbols(q_m)$. If such an element t, called a *merge candidate*, is found, a new configuration is obtained by temporarily replacing t_1 and t_2 with t in the input sequence and the parse_check distance is computed for this new configuration.

As described in the previous section, the deferred driver insures that the state q_m on top of the stack of the error configuration is the state entered prior to the execution of any action on t_1. In that configuration, it may be possible to execute a sequence of reduce, goto-reduce and goto actions before the illegality of t_1 is detected in another state q_e. In such a case, the elements in $t\_symbols(q_m)$ that are also in $t\_symbols(q_e)$ are given priority in applying the insertion and substitution trials. (It is not hard to show that $t\_symbols(q_e) \subseteq$ t_symbols(q_m).) The benefits of this ordering can be seen in the following example:

```
write(1*5+6;2*3,4/2)
```

In this erroneous Pascal statement, a semicolon is used instead of a comma after the first parameter. Assume state q_m is the first state that encounters the semicolon. At that point, the parser has just shifted an expression operand and the set of valid lookahead symbols includes not only the comma but all the arithmetic operators. However, if the parser is allowed to interpret the operand as a complete expression, it will enter an error state q_e where the comma is the only candidate.

In order to give priority to the candidates in an error state q_e, it is necessary to identify when the parser has entered such a state. State q_e can be computed in the *lookahead_action* function by inserting the following statement after lines 3. and 13. in Figure 3:

```
error_state := act;
```

3.1.1 The Mispelling Index

For a successful merge trial, the misspelling index is set to 1.0 since the merged string must perfectly match the character string representation of the merge candidate.

As mentioned earlier, a misspelling function is invoked to calculate the misspelling index for a simple substitution. The misspelling function used in this method was proposed by Uhl [14]. The distance between two words is measured by the number of letter inversions, insertions and deletions. The smaller the distance between two words, the

more likely it is that one is a misspelling of the other.

For all other recoveries, the misspelling index is set to 0.0.

3.2 Secondary Recovery

Secondary recovery (also called *Phrase-level recovery* [8] [12]) is based on the identification of an *error phrase* which is then deleted from the input or replaced by a suitable nonterminal symbol or *reduction goal*. If the string:

$$q_1, \ldots, q_m \quad | \quad t_1, \ldots, t_n \qquad (1)$$

is an error configuration, then a substring

$$q_{i+1}, \ldots, q_m \quad | \quad t_1, \ldots, t_{j-1} \qquad (2)$$

$1 \leq i \leq m$, $1 \leq j \leq n$, of that configuration is an error phrase (of the configuration) if removing that substring allows the parser to advance at least *min_distance* tokens into the forward context, or if there is a nonterminal A such that a valid action is defined in state q_i on A, and after processing A, the parser can advance at least *min_distance* into the forward context. Here, q_i, A and t_j are the *recovery state*, *reduction goal* and *recovery symbol*, respectively.

The scheme used in this method to select error phrases reflects a fundamental distinction that is made among three different kinds of errors. Consider the error configuration (2) above. The case of the empty error phrase is considered during primary recovery as a nonterminal insertion. Similarly, the case where an error phrase $\epsilon|t_1$ is deleted or replaced by a nonterminal candidate is processed by a primary recovery deletion or nonterminal substitution. Next, priority is given to a successful secondary recovery that consumes no input symbol and requires no insertion of a reduction goal; i.e., a recovery based on the removal of an error phrase of the form $\beta|\varepsilon$ where $\beta \neq \epsilon$. This kind of error is called a *misplacement error*, and β is called a *misplaced phrase*. The following Pascal program illustrates this case:

```
 1. program P(INPUT,OUTPUT);
 2.    var I:real;
       <--------->
*Error: Misplaced construct(s)
 3.    type ORDER=array[1..MAX] of real;
 4.    var Q:integer;
 5. begin
 6. end.
```

Finally, the case in which one or more input symbols and/or states must be deleted or replaced with a nonterminal candidate is considered. In that case, input symbols are consumed faster than states. In other words, the error phrases are selected as indicated by the row-major order of the table below:

$$\begin{array}{ccc} \varepsilon|\varepsilon & \cdots & \varepsilon|t_1, \ldots, t_n \\ q_m|\varepsilon & \cdots & q_m|t_1, \ldots, t_n \\ \vdots & & \vdots \\ q_2, \ldots, q_m|\varepsilon & \cdots & q_2, \ldots, q_m|t_1, \ldots, t_n \end{array}$$

In this final case, each error phrase selected is removed from the base configuration (1). An initial attempt is made to recover by parse checking the resulting configuration. This action, called *secondary deletion*, can be viewed as a multiple deletion of the symbols that make up the error phrase. Next, each element in the set of nonterminal candidates for the newly exposed state on top of the state stack is substituted, in turn, for the error phrase and the *parse_check* function is invoked to determine its viability. This action is called a *secondary substitution*. This process continues until a successful recovery is found or all the possibilities are exhausted.

In secondary recovery, the aim is to find a repair that least alters the original configuration. For this reason, misplacement trials are performed separately from the other secondary trials and given higher priority, since such a repair does not delete any symbol from the forward context and tends to remove whole structures from the left context that have been previously analysed. The *parse_check* distance is used as the criterion to select the best misplacement repair. After the misplacement trials, a secondary deletion and substitution trial is performed on successive error phrases. The selection of a best deletion or substitution repair is based on the length of the relevant error phrase and the *parse_check* distance, with deletion having priority over substitution in case of a tie. The length of an error phrase $\beta|x$ is obtained by adding the length of the string x to the number of non-null symbols in β.

Given the best misplacement repair and the best deletion or substitution repair, if the misplacement repair is based on a shorter error phrase or it yields a longer *parse_check* distance, then it is chosen. Otherwise, the deletion or substitution is chosen.

3.3 Optimization of Candidates

Consider the case of a secondary substitution in which a recovery goal A must be inserted into the input stream. In such a case, every nonterminal

$E \rightarrow \cdot E + T$	$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$	$T \rightarrow \cdot F$
$F \rightarrow \cdot F \uparrow P$	$F \rightarrow \cdot P$
$P \rightarrow \cdot id$	$P \rightarrow \cdot (E)$

Figure 5: Items in a state q_i

candidate in state q_i is a potential reduction goal. However, an implementation that checks all potential candidates for each error phrase would be prohibitively slow.

Two optimizations are applied to the set of nonterminal candidates in a given state to obtain, in most cases, a substantially reduced subset of *relevant reduction goals.*

In [8], the following concept is presented: a reduction goal A of error phrase $\beta|x$ in error configuration $\alpha\beta|xy$ is *important* if $\beta|x$ has no reduction goal B such that $B \rightarrow^+ A$. In this method a more restricted concept of an important symbol is used. The new concept takes into consideration the full context of the error phrase. A nonterminal A on which a transition is defined in a state q_i is said to be *important* if A does not appear in a single item of the form $B \rightarrow \cdot A$ in q_i. For example, assume a recovery state q_i contains the set of items shown in Figure 5. By the definition of [8], the only important reduction goal in such a state is E, since T, F and P can be derived from E via a chain of unit productions. By the more restricted definition of this method, T and F would also be considered important symbols since they appear immediately to the right of the dot in more than one item. To understand the importance of T and F, assume that the rules from which the items of Figure 5 are derived are all the productions of a grammar and consider the following erroneous input strings:

```
( ) ) ( * id + id
( ) ) ( ↑ id + id
```

If E is the only important symbol considered, then the best secondary repair that is achievable is the replacement of "`( ) ) ( * id`" by E in the first sentence and "`( ) ) ( ↑ id`" by E in the second sentence. However, it is clear from the grammar that replacing "`( ) )` (" by T in the first sentence and by F in the second sentence would be preferable.

One further notices that using F as a reduction goal in the first sentence would have worked just as well, since after a transition on F, with the symbol "*" as lookahead, a reduction by the rule "$T \rightarrow F$" would be applied. Similarly, P could have been used as a suitable reduction goal in both sentences. This leads to the following concept, on which the second optimization is based: a nonterminal element C of a set of nonterminal candidates S in an LR state q is said to be *relevant* with respect to S if there does not exist a nonterminal D, such that $D \in S$, $D \neq C$, and D can be successfully substituted for C as a reduction goal for any error phrase with q as the recovery state.

Given a set S of nonterminal candidates for a given state, the objective is to find the largest subset $S' \subseteq S$ such that S' contains only relevant reduction goals. Let $S = \{B_1, \ldots, B_k\}$ for $1 \leq i \leq k$, $B_i \in S$ is relevant iff $\nexists B_j$, $j \neq i$, such that $B_i \Rightarrow^+_{rm} B_j$. The proof of this assertion follows directly from the definition of an LR parser. If a nonterminal B can be substituted for an error phrase, then the recovery symbol t in question must be a valid lookahead symbol for any rule derivable from B. In particular, if $B_i \Rightarrow^+_{rm} B_j$ and B_j is substituted for an error phase where B_i is known to be a valid reduction goal, the recovery symbol will cause B_j to be reduced to B_i.

For each state q in an LR automaton, the set *nt_symbols*(q) is obtained as follows. Starting with the set of nonterminal symbols on which an action is defined in q, remove all unimportant symbols from that set, and reduce the resulting set further by removing all irrelevant reduction goals from it. For example, consider the state q_i of Figure 5. State q_i contains nonterminal transitions on the symbols E, T, F and P. The only unimportant symbol in that set is P. After P is removed, the irrelevant symbols E and T are removed from the subset $\{E, T, F\}$ leaving F as the only relevant reduction goal in q_i.

The notion of an important symbol can also be extended to terminal candidates in the *t_symbols* sets. Once again, consider the state q_i of Figure 5. This state contains a single terminal action on the symbol *id*, but, since *id* appears only in the item $P \rightarrow \cdot id$, it is not an important candidate in q_i. The removal of unimportant terminals improves the time performance of the primary recovery and saves space. However, it may suppress some opportunities for merging and misspelling corrections.

In [13], an algorithm is presented that can be used to further reduce the space used by *t_symbols* and *nt_symbols*.

3.4 Scope Recovery

One of the most common errors committed by programmers is the omission of block closers such as an *end* statement or a right parenthesis. Such

```
if_stmt → IF cond THEN
            st_list  elsif_list  opt_else
          END IF ;
st_list → stmt  |  st_list stmt
elsif_list → ε |  elsif_list ELSIF cond THEN st_list
opt_else → ε |  ELSE st_list
stmt → ... |  if_stmt  |  ...
```

Figure 6: BNF rule for Ada **if** statement

an error is referred to as a *scope error*. Scope errors are common because the structures requiring block closers are usually recursive structures that, in practice, are specified in a nested fashion. In such a case, a matching block closer must accompany each structure in the nest. For example, if a user specifies an expression that is missing a single right parenthesis, primary recovery can successfully insert that symbol. However, if two or more right parenthesis are missing, neither primary nor secondary recovery can successfully repair such an error. Similarly, consider the BNF rule for an Ada *if statement* in Figure 6 [9]: If an Ada *if statement* is specified without the "END IF ;" closer, neither of the two recovery techniques mentioned so far can effectively repair this error. The repair that is necessary for this kind of error is the insertion of a sequence of symbols, called *multiple symbol insertion.*

Scope recovery was first introduced by Burke and Fisher [11]. Their technique requires that each closing sequence be supplied by the user as a list of terminal symbols. Scope recovery is attempted by checking whether or not the insertion of a combination of these closing sequences can allow the parser to recover.

By contrast, the scope recovery technique used in this method is based on the identification of one or more recursively defined rules that are incompletely specified, and insertion of the appropriate closing symbols to complete these phrases. All necessary scope information required by this method is precomputed automatically from the input grammar. In addition, the method is based on a pattern match with complete rules rather than just the insertion of closing sequences of terminal symbols. As a result, the diagnosis of scope errors is more accurate in that it identifies whole structures that are incompletely specified instead of just the missing sequence of closing terminals.

3.4.1 Scope Information

Definition 3.1 *A rule* $A \rightarrow \alpha B \beta$ *is a* **scoped rule** *if* $\alpha \neq \varepsilon$, $B \Rightarrow^* \gamma A \delta$, *for some arbitrary string* γ *and* δ, *and* $\beta \not\Rightarrow^* \epsilon$.

In the example of Figure 5, the rule $P \rightarrow (E)$ is a scoped rule since P can be derived from E. The **if_stmt** rule of Figure 6 is also a scoped rule since each of the bold symbols following **THEN** in that rule can recursively derive a string containing the symbol **if_stmt**. A *scope* can be derived from a scoped rule for each recursive symbol in the right-hand side of the scoped rule.

A scope is a quintuple (π, σ, a, A, Q) where π and σ are strings of symbols called *scope prefix* and *scope suffix*, respectively, a is a terminal symbol called the *scope lookahead*, A is a nonterminal symbol called the *left-hand side* and Q is a set of states. The scope prefix is the prefix of a *suitable string* derivable from the scoped rule in question. It is used to determine whether or not a recovery by the associated scope is applicable; i.e., at run time, a repair by a given scope is considered only if this initial substring of the suitable string can be successfully derived before the error token causes an error action. The scope suffix is the suffix (of the suitable string) that follows the scope prefix. When diagnosing a scope error, the user is advised to insert the symbols of the scope suffix into the input stream to complete the specification of the scoped rule. The scope lookahead symbol (string, if the grammar is LR(k)) is a terminal symbol (string) that may immediately follow the prefix in a legal input. The left-hand side of the scope is the nonterminal on the left of the scoped rule. The set Q contains the states of the LR(k) automaton in which the left-hand side can be introduced through closure.

Given a scoped rule $A \rightarrow \alpha B \beta$, the scope information related to B is computed as follows. Since $\beta \not\Rightarrow^* \epsilon$, there exists a string $\psi X \phi$ such that $\beta \Rightarrow^* \psi X \phi$, $\psi \Rightarrow^* \epsilon$, and $X \Rightarrow^*_{rm} a\omega$. Let $\alpha B \psi X \phi$ be the suitable string mentioned above, then a valid *scope* for the above rule is $(\alpha B \psi, X \phi, a, A, Q)$, where Q is the set of states in the LR automaton containing a transition on A.

As an example, consider the **if_stmt** rule of Figure 6 and the scope induced by the nonterminal **st_list** in its right-hand side. To put it in the form $A \rightarrow \alpha B \beta$, let B be the symbol "**st_list**". It follows that α is the string "IF cond THEN", and β is the string "**elsif_list opt_else** END IF ;". Let ψ be the string "**elsif_list opt_else**" and let X be the symbol "END". One observes that β is exactly in

```
# Let scope_seq be a global output variable.
# Initially, scope_seq= [ ] and scope_trial is
# invoked with the sequence q_1, ..., q_m. The input
# sequence t_1, ..., t_n is assumed to be global.
proc scope_trial(stack);
{ for each scope (π_i, σ_i, a_i, A_i, Q_i) do
  { sstk := stack;
    act := lookahead_action(sstk, a_i, pos)
    if act ≠ error then
    { sstk[pos+1..] := tstk[pos+1..];
      top := #sstk - |π_i|;
      if top > 0 then
      { pref := [in_sym[sstk[j]] : j in top+1..#sstk];
        if pref = π_i and sstk[top] ∈ Q_i then
        { do
          { top := top - rhs[act] + 1;
            act := GOTO(sstk[top], lhs[act]);
          } while act is a goto-reduce action
          sstk[top+1 ..] := [act];
          if prschck(sstk, t_1, ..., t_n) > min_dist then
          { scope_seq := [i];
            return;
          }
          else
          { scope_trial(sstk);
            if scope_seq ≠ [ ] then
              scope_seq := scope_seq + [i];
            return;
          }
        }
      }
    }
  }
}
```

Figure 7: *scope_trial* procedure

the desired form $\psi X \phi$. Thus, assuming the set of transition states Q is available, the scope induced by **st_list** for the rule if_stmt is:

(IF cond THEN st_list elsif_list opt_else, END IF ;, END, if_stmt, Q)

The other recursive symbols in if_stmt: **elsif_list** and **opt_else** induce exactly the same scope as **st_list**, since they are both *nullable*.

3.4.2 Scope Error Detection

Given an error configuration:

$$q_1, \ldots, q_m \quad | \quad t_1, \ldots, t_n$$

and a set of scopes:

$$\{(\pi_1, \sigma_1, a_1, A_1, Q_1), \ \ldots, \ (\pi_l, \sigma_l, a_l, A_l, Q_l)\},$$

the applicability of scope recovery to this configuration is determined as follows. For each scope $(\pi_i, \sigma_i, a_i, A_i, Q_i)$, a three-step test is performed:

step 1: The *lookahead_action* function is invoked with a_i as the current token to check if a_i is a valid lookahead symbol for the viable prefix. As a side-effect, this function updates the state stack configuration (using a temporary stack) to reflect all reduce actions, including empty reductions, induced by a_i. If the action returned by *lookaheadaction* is the error action then the whole test fails. Otherwise, step 2 is executed.

step 2: A pattern match is made between the prefix π_i and the topmost $|\pi_i|$ symbols of the viable prefix, i.e., the string obtained from the concatenation of the in_symbols of the states: $q_{m-|\pi_i|+1} \cdots q_m$. Again, if this test fails, the whole test fails. Otherwise the final step is executed.

step 3: If $q_{m-|\pi_i|} \in Q_i$ then the test is successful. Otherwise, the test fails.

If the three-step test is successful, then a parse check is performed on the configuration: $q_1, \ldots, q_{m-|\pi_i|}, q_A \mid t_1, \ldots t_n$, where q_A is the successor state of q_m and A[1]. If the *parse_check* function can parse at least *min_distance* symbols, the scope recovery is successful. Otherwise, it is invoked recursively with the new configuration above and the process is repeated until scope recovery either succeeds, or there are no more possibilities to try.

When scope recovery is successful, the sequence of scopes that resulted in the successful recovery must be saved for the issuance of an accurate diagnostic.

Figure 7 shows a complete implementation of the scope error detection algorithm. The algorithm mirrors the preceding discussion in a straightforward manner. The emphasis in writing the code was on the clarity of the exposition rather than efficiency.

4 Recovery Phases

This section describes how the different repair strategies discussed in the previous sections are in-

[1] If the action in q_m on A is a goto-reduce, the parser is simulated through the whole sequence of goto-reduce actions that follow, until a goto action is encountered. This final goto is executed and the resulting state sequence is used instead. Note that these actions do not consume any input symbol.

corporated into the unified two-phase scheme of this method. At the global level, the effectiveness of a recovery trial is measured based on two criteria:

- the number of symbols that must be deleted if the repair in question is applied
- the *parse_check* distance of the recovery

The primary phase recovery which includes all recovery trials that are based on at most a single input token modification is attempted first. If a successful primary phase recovery is found that cannot be beaten by any other recovery in terms of the criteria above, it is accepted. If such a primary phase recovery is not found, secondary phase recovery is attempted. If a successful secondary phase recovery is found, then it is accepted. Otherwise, the error recovery gets into a form of panic mode, where the current input buffer is flushed, new input tokens are read in and secondary phase recovery is attempted again. This process is repeated until either a successful secondary recovery is obtained or the end of the input stream is reached.

When a recovery is accepted, the following actions are taken: a diagnosis is issued, the repair is applied and the error recovery procedure returns successfully.

The diagnosis of a primary recovery is straightforward. To diagnose a secondary deletion, the user is advised to delete the symbols in the error phrase in question. Similarly, for a secondary substitution, the relevant reduction goal is suggested as a replacement for the error phrase. The location of an error phrase starts from the location associated with the recovery state to the location of the last token in the error phrase. To diagnose a scope recovery, the location of *prevtok* is used to indicate where the symbols of the scope suffix in question should be inserted.

A repair is applied by resetting the components of the main configuration (*buffer* and *stk*). The resetting of the input buffer simply involves the insertion of some symbols into the buffer, the reading of new input tokens into the buffer, or the replacement of some buffer elements. The resetting of the stack is more complicated. For a primary recovery, one only needs to choose the stack on which the recovery was successful. For a secondary recovery, all states following the recovery state are removed from the stack. For a scope recovery, the sequence of states on top of the stack that corresponds to the prefix of the scope is removed and the repair proceeds as if the error was a simple insertion of the left-hand side of the scope.

4.0.3 Primary Phase

In the primary phase, error recovery is applied on each available configuration, starting with *nstk*, proceeding with *stk* and finally processing *pstk*. For each configuration, scope recovery is attempted first followed by primary recovery. The same criteria used in choosing a primary recovery is used in the primary phase. The misspelling index of a scope recovery trial is set to 1.0. Thus, for a given configuration, a successful scope recovery always has priority over a primary recovery trial that yields the same parse_check distance.

If a successful recovery is obtained from the primary phase and its stack configuration is *nstk* or *stk*, the recovery trial is evaluated against certain secondary recovery trials on the stack configuration in question before being accepted. These recovery trials are the ones whose repair actions would have as little impact on the recovery configuration as a primary recovery. They are misplacement recovery trials and scope recovery trials that require the deletion of one input token. The idea is to ensure that none of these borderline recoveries can be more effective than the best primary phase recovery.

4.0.4 Secondary Phase

In the secondary phase, secondary error recovery is applied first on *nstk* if it is available and then on *stk*. If a successful secondary recovery is obtained, a check is made to see if the error can be better repaired by the closing of some scopes followed by less radical surgery. Consider the following Pascal example:

```
if count[listdata[sub] := 0 then
    x := (( 3 ]];
```

In the first line, the user is missing a closing "]" and the assignment operator ":=" is used instead of a relational operator. This error is detected on the symbol ":=". In the second line, the user used the wrong closing symbols in an expression and the error is detected on the first "]". Nothing short of a secondary deletion of the sequence "`[listdata[sub] := 0`" in the first instance and a secondary substitution of "expression" for the sequence "`(( 3 ]]`" would successfully repair these errors. However, it is not difficult to see that they can be repaired more accurately, using scope recovery by proceeding as follows.

Before accepting a secondary recovery based on an error phrase $\beta|x$, a scope recovery check is performed on the recovery configuration, followed by the deletion of up to $|x|$ tokens in the right context. If the scope recovery is successful, then its

associated repair actions are applied without the subsequent deletion and the secondary phase returns successfully. The parser fails right away and once again invokes the error recovery procedure. On this next round, primary and secondary phase recovery are attempted again. This subsequent attempt will at best fix the remaining input or at worst delete a string up to the length x from the input. In the example above, the missing "]" is inserted and "relational_operator" is substituted for ":=" in the first line. In the second line, two closing ")" are inserted, followed by a deletion of the pair "]]" (See figure 2).

5 Implementation

The error recovery method described in this paper has been successfully implemented. An LALR(k) parser generator was modified to produce the extra tables required: *t_symbols*, *nt_symbols* and the scopes. The method can be used with any LR(k) application. However, programming languages were used in our examples because such applications are the best illustrations of the problems one is likely to encounter. Parsers were built for Ada and Pascal and tested on the Ada examples of [11] and the Pascal examples of [6].

Penello and DeRemer [4] proposed that the quality of a repair be rated "excellent" if it repaired the test as a human reader would have, "good" if not but it still resulted in a reasonable program and no spurious errors, and "poor" if it resulted in one or more spurious errors. Based on these categories, the performance of this method on the test set of [6] was 85.9% excellent, 14.1% good and 0.0% poor. In fact, most of the "good" recoveries resulted from errors whose repair required some kind of semantic judgement.

The time performance of this method is excellent, usually requiring less than 50 milliseconds per error on a 16 MHz PS/2 model 80.

6 Conclusion

This paper described a new practical LR(k) error diagnosis and recovery method which improves upon the current state-of-the-art in some significant ways. Specifically,

- a new deferred driver is introduced which always detects an error at the earliest possible point;
- the primary recovery is generalized to process both terminal and nonterminal symbols;
- the secondary recovery is an efficient (and completely automatic) generalization of the error production method;
- techniques are presented for optimizing error recovery candidates;
- a new automatic method for scope recovery is presented.

Moreover, this method is completely language- and machine-independent and more efficient than other known methods.

7 Acknowledgements

The author wishes to thank the following people for many helpful suggestions and their encouragement throughout the development of this work: Michael Burke, Ron Cytron, Gerald Fisher, Laurent Pautet, Matthew Smosna. The author is especially thankful to Fran Allen and Ed Schonberg for their advice and support.

References

[1] F. L. DeRemer
Practical Translators for LR(k) Languages.
Ph.D. dissertation, MIT, Cambridge, Mass., 1969

[2] F. L. DeRemer: Simple LR(k) Grammars.
Comm. ACM 14, 7, 453-460 July 1971

[3] Alfred V. Aho, Jeffrey D. Ullman
The Theory of Parsing, Translation, and Compiling
Volume I & II, Prentice Hall, Inc 1972

[4] Penello, T. J., and DeRemer, F. L.
A forward move algorithm for LR error recovery.
ACM Symposium on Principles of Programming Languages (Jan. 23-25, 1978, Tuscon), pp. 241-254

[5] Ripley, G. D., and Druseikis, F.C.
A statistical analysis of syntax errors
Journal of Computer Languages 3,4 (1978) (227-240)

[6] Ripley, D. J.: Pascal Syntax Errors Data Base
RCA Laboratories, Princeton, N.J., Apr 1979

[7] S. L. Graham, C. B. Haley, W. N. Joy
Practical LR Error Recovery
SIGPLAN 79 Symposium on Compiler Construction
(August 6-10, 1979, Denver) ACM, NY, pp 168-175.

[8] Seppo Sippu, Eljas Soisalon-Soininen
A Syntax-Error-Handling Technique and Its Experimental Analysis
ACM Transactions on Programming Languages and Systems, Vol. 5, No. 4, October 1983, Pages 656-679

[9] Ref. Manual for the ADA Programming Language
ANSI/Mil-STD-1815A-1983, U.S. Dept. of Defense.

[10] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Compilers: Principles, Techniques and Tools
Addison Wesley Publishing Company, 1986

[11] Michael Burke, Gerald A. Fisher
A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery
ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2, April 1987, Pages 164-197

[12] Nigel P. Chapman: LR Parsing: Theory and Practice
Cambridge University Press, 1987

[13] Philippe Charles, Laurent Pautet
Efficient Representation of LR Error Recovery tables
Unpublished paper, 1989

[14] Jüergen Uhl: Private communications

ADAPTIVE PROBABILISTIC GENERALIZED LR PARSING

Jerry Wright*, **Ave Wrigley*** and **Richard Sharman**+
* *Centre for Communications Research*
Queen's Building, University Walk, Bristol BS8 1TR, U.K.
+ *I.B.M. United Kingdom Scientific Centre*
Athelstan House, St Clement Street, Winchester SO23 9DR, U.K.

ABSTRACT

Various issues in the implementation of generalized LR parsing with probability are discussed. A method for preventing the generation of infinite numbers of states is described and the space requirements of the parsing tables are assessed for a substantial natural-language grammar. Because of a high degree of ambiguity in the grammar, there are many multiple entries and the tables are rather large. A new method for grammar adaptation is introduced which may help to reduce this problem. A probabilistic version of the Tomita parse forest is also described.

1. INTRODUCTION

The generalized LR parsing algorithm of Tomita (1986) allows most context-free grammars to be parsed with high efficiency. For applications in speech recognition (and perhaps elsewhere) there is a need for a systematic treatment of uncertainty in language modelling, pattern recognition and parsing. Probabilistic grammars are increasing in importance as language models (Sharman, 1989, Lari and Young, 1990), pattern recognition is guided by predictions of forthcoming words (one application of the recent algorithm of Jelinek (1990)), and the extension of the Tomita algorithm to probabilistic grammars (Wright *et al*, 1989, 1990) is one approach to the parsing problem. The successful application of the Viterbi beam-search algorithm to connected speech recognition (Lee, 1989), together with the possibility of building grammar-level modelling into this framework (Lee and Rabiner, 1989) is further evidence of this trend. The purpose of this paper is to consider some issues in the implementation of probabilistic generalized LR parsing.

The objectives of our current work on language modelling and parsing for speech recognition can be summarised as follows:

(1) real-time parsing without excessive space requirements,

(2) minimum restrictions on the grammar (ambiguity, null rules, left-recursion all permitted, no need to use a normal form),

(3) probabilistic predictions to be made available to the pattern matcher, with word or phoneme likelihoods received in return,

(4) interpretations, ranked by overall probability, to be made available to the user,

(5) adaptation of the language model and parser, with minimum delay and interaction with the user.

The choice of parser generator is relevant to objectives (1) and (3). All versions satisfy objective (2) but are initially susceptible to generating an infinite number of states for certain probabilistic grammars, and in the case of the canonical parser generator this can happen in two ways. A solution to this problem is described in section 2. The need for probabilistic predictions of forthcoming words or phonemes (objective (3)) is best met by the canonical parser generator,

because for all other versions the prior probability distribution can only be found by following up all possible reduce actions in a state, in advance of the next input (Wright *et al*, 1989, 1990). This consumes both time and space, but the size of the parsing tables produced by the canonical parser generator generally precludes their use. The space requirements for the various parser generators are assessed in section 3. The grammar used for this purpose was developed by I.B.M. from an Associated Press corpus of text (Sharman, 1989).

Objective (4) is met by the parse forest representation which is a probabilistic version of that employed by Tomita (1986), incorporating sub-node sharing and local ambiguity packing. This is described in section 4.

The final issue (objective (5) and section 5) is crucial to the applicability of the whole approach. We regard a grammar as a probabilistic structured hierarchical model of language as used, not a prescriptive basis for correctness of that use. A relatively compact parsing table presumes a relatively compact grammar, which is therefore going to be inadequate to cope with the range of usage to which it is likely to be exposed. It is essential that the software be made adaptive, and our experimental version operates through the LR parser to synthesise new grammar rules, assess their plausibility, and make incremental changes to the LR parsing tables in order to add or delete rules in the grammar.

2. PARSING TABLE SPACE REQUIREMENTS

2.1 INFINITE SERIES OF STATES: FROM THE ITEM PROBABILITIES

When applied to a probabilistic grammar, the various versions of LR parser generator first produce a series of item sets in which a probability derived from the grammar is attached to each item, and then generate the *action* and *goto* tables in which each entry again has an attached probability, representing a frequency for that action conditional upon the state (Wright *et al*, 1989, 1990). Sometimes these probabilities can cause a problem in state generation. For example, consider the following probabilistic grammar:

$$S \rightarrow A,\ p_1 \mid B,\ p_2$$

$$A \rightarrow c\ A,\ q_1 \mid a,\ q_2$$

$$B \rightarrow c\ B,\ r_1 \mid b,\ r_2$$

where p_1, p_2 and so on (with $p_1 + p_2 = 1$) represent the probabilities of the respective rules. After receiving the terminal symbol c, the state with the (closed) item set shown in Table 1 is entered, with the first column of probabilities for each item. After receiving the terminal symbol c again, a state with the same item set is entered but with the second column of probabilities for each item, and these are different from the first unless $q_1 = r_1$. For the probabilistic parser these states must therefore be distinguished, and in fact this process continues to generate an (in principle) infinite sequence of states. Although it may sometimes be sufficient merely to truncate this series at some point, the number of additional states generated when all the "goto" steps have been exhausted can be very large.

Table 1: Item set with probabilities.

Item	First probability	Second probability
$A \to c \cdot A$	$p_1 q_1/(p_1 q_1+p_2 r_1)$	$p_1 q_1^2/(p_1 q_1^2+p_2 r_1^2)$
$B \to c \cdot B$	$p_2 r_1/(p_1 q_1+p_2 r_1)$	$p_2 r_1^2/(p_1 q_1^2+p_2 r_1^2)$
$A \to \cdot\ c\ A$	$p_1 q_1^2/(p_1 q_1+p_2 r_1)$	$p_1 q_1^3/(p_1 q_1^2+p_2 r_1^2)$
$A \to \cdot\ a$	$p_1 q_1 q_2/(p_1 q_1+p_2 r_1)$	$p_1 q_1^2 q_2/(p_1 q_1^2+p_2 r_1^2)$
$B \to \cdot\ c\ B$	$p_2 r_1^2/(p_1 q_1+p_2 r_1)$	$p_2 r_1^3/(p_1 q_1^2+p_2 r_1^2)$
$B \to \cdot\ b$	$p_2 r_1 r_2/(p_1 q_1+p_2 r_1)$	$p_2 r_1^2 r_2/(p_1 q_1^2+p_2 r_1^2)$

Table 2: Separated item sets.

$A \to c \cdot A$	1
$A \to \cdot\ c\ A$	q_1
$A \to \cdot\ a$	q_2

$B \to c \cdot B$	1
$B \to \cdot\ c\ B$	r_1
$B \to \cdot\ b$	r_2

We can avoid this problem by introducing a multiple shift entry for the terminal symbol c, in the state from which the one just discussed is entered. Multiple entries in the *action* table are normally confined to cases of shift-reduce and reduce-reduce conflicts, but the purpose here is to force the stack to divide, with a probability $p_1 q_1/(p_1 q_1+p_2 r_1)$ attached to one branch and $p_2 r_1/(p_1 q_1+p_2 r_1)$ to the other. These then lead to separate states with item sets as shown in Table 2.

The prior probabilities of a, b and c are obtained by combining the two branches and take the same values as before. Further occurrences of the terminal symbol c simply cause the same state to be re-entered in each branch, and eventually an a or b eliminates one branch. If c is replaced by a nonterminal symbol C, the same procedure applies except that a multiple *goto* entry is required, and this in turn means that a probability has to be attached to each *goto* entry (this was not required in the original version of the probabilistic LR parser).

Suppose in general that a grammar has nonterminal and terminal vocabularies N and T respectively. Conditions for the occurrence of an infinite series of states can be summarised as follows: there occurs a state in the closure of which there arise either

(a) two distinct self-recursive nonterminal symbols (A, B say) for which a nonempty string $\alpha \in (N \cup T)^+$ exists such that

$$A \overset{*}{\Rightarrow} \alpha\ A\ \beta \quad \text{and} \quad B \overset{*}{\Rightarrow} \alpha\ B\ \gamma$$

where $\beta, \gamma \in (N \cup T)^*$,

or (b) two (or more) mutually recursive nonterminal symbols for which a nonempty string $\alpha \in (N \cup T)^+$ exists such that

$$A \overset{*}{\Rightarrow} \alpha\ B\ \beta \quad \text{and} \quad B \overset{*}{\Rightarrow} \alpha\ A\ \gamma$$

where $\beta,\gamma \in (N \cup T)^*$, and in addition either

(i) a left-most derivation of one symbol from the other is possible:

$$A \overset{*}{\Rightarrow} B\ \delta \quad \text{where } \delta \in (N \cup T)^*$$

or (ii) a self-recursive nonterminal (C, say, which may coincide with A or B) also arises such that

$$C \overset{*}{\Rightarrow} \alpha\ C\ \theta \quad \text{where } \theta \in (N \cup T)^*$$

for the same α.

These conditions ensure that the α-successor of this state also contains items with A and B after the dot but with probabilities different from those for the earlier state, and moreover that this continues to generate an infinite series (different item probabilities do not always imply this). One way to prevent this series is to associate with each item in the lists (which form the states) an array of states for each nonterminal symbol, recording the state(s) in which that symbol occurred as the left-hand side of an item from which the current item is descended. These arrays can be created during the course of the LR "closure" function. Pairs of nonterminals satisfying the conditions above are then easily detected within the "goto" function, so that an appropriate multiple shift or goto entry can be automatically created for the last symbol of α. Only the items leading to the looping behaviour need to be separated by this means, and the number of additional states generated is small. Cases of three-way (or higher) mutual recursion with a common α are very rare.

2.2 INFINITE SERIES OF STATES: FROM THE LOOKAHEAD DISTRIBUTION

For the probabilistic LALR parser generator the lookaheads consist of a set of terminal symbols, as in the case of non-probabilistic grammars. However, for the canonical parser generator there is a full probability distribution of lookaheads and this creates a second potential source of looping behaviour. It is possible for item sets to have the same item probabilities but different lookahead distributions. Suppose that a state contains an item with a right-recursive nonterminal symbol (C, say) after the dot, with nothing following. If the state also contains another item with C after the dot followed by a non-null string, thus

$$C \overset{*}{\Rightarrow} \alpha\ C \quad \text{and} \quad C \overset{*}{\Rightarrow} \alpha\ C\ \beta$$

where $\alpha,\beta \in (N \cup T)^+$, then the new lookahead probability distribution computed for C will be a mixture of the old one and a distribution derived from β in the second item. The state automaton possesses a loop because of the right-recursion, and the lookahead distribution is different each time around so that again an (in principle) infinite series is generated. The second item can arise within the same state if C is also left-recursive (the simplest example of this is the grammar $S \rightarrow S\ S,\ p_1 \mid a,\ p_2$), and this problem also arises for an item of the second kind on its own, if β is nonempty but nullable.

It is possible to break the loop by introducing multiple shift (or goto) actions as before, but the procedure is complicated by the presence of null rules and/or left-recursion. These can allow the distribution to change even when there is just a single item in the kernel. In the absence of this behaviour the state from which the one just discussed is entered can be

treated with a multiple entry in order to prevent the lookaheads from mixing, the conditions which give rise to this problem being checked within the "goto" function. The prior probability calculations at run-time are correct.

This procedure has not been fully implemented at the present time, but it seems that this kind of looping behaviour is more common than that discussed in the previous section. The additional states created by the multiple shifts exacerbates the already major disadvantage of the canonical parser with regard to space requirements.

2.3 MERGING OF CANONICAL STATES

Consider the following grammar:

$S \rightarrow A \mid b\ A\ c$

$A \rightarrow e\ f\ S \mid g$

(the rule-probabilities do not matter). The full canonical parser generator produces eighteen states, of which eight are eliminated by shift-reduce optimisation (Aho *et al*, 1985). Of the remaining states, a further eight consist of two sets of four, the sets distinguished only by the lookaheads. These states propagate the dot through the longer rules, but in fact the lookaheads are not used because in each case the series terminates in a shift-reduce entry. When this action occurs the parser moves to a state wherein the possible next symbols are revealed. These states can therefore be merged without compromising the predictive advantage of the canonical parser. This reduces the number of states to six, the same as for the LALR parser generator.

All this applies to the probabilistic version where the lookaheads are propagated as a distribution. A fairly simple procedure allows each state to be endowed with a flag to indicate whether or not any of the lookahead data are important. If not, merger can be based purely on the rule and dot positions for the items, and their probabilities. The numbers of states saved varies very much with the grammar: the above example represents an extreme case, and equally there are grammars for which no saving occurs.

3. COMPARISON OF PARSER GENERATORS

To compare the parser generators a test grammar developed by I.B.M. from an Associated Press corpus was used (Sharman, 1989). This grammar consists of 677 rules, ranked with a rule-count which was easily converted into a probability. It was convenient to use reduced versions of the grammar based on a rule-count threshold. Simply truncating the grammar is not sufficient, however, for two reasons. First, the resulting grammar can be disconnected in that there exist rules whose left-hand sides cannot occur in any string derived from S. Second, the grammar can be incomplete in that nonterminal symbols can arise within strings derived from S but for which there are no corresponding rules because all have counts below the threshold. The solution to these two problems is basically the same: recursively to add to the truncated grammar a small number of additional rules, with counts below the threshold, until the resulting grammar is connected and complete.

Applying this procedure for various rule-count thresholds creates a hierarchy of grammars and allows the relationship between the size of the grammar and the parsing tables to be explored. Table 3 contains a summary of the results. The number of states and total

Table 3: Parsing table space requirements.

Rules	Size	Non-prob LALR States	Non-prob LALR Entries	Non-prob LALR %>1	Probabilistic LALR States	Probabilistic LALR Entries	Probabilistic LALR %>1	Canonical States	Canonical Entries	Canonical %>1
15	37	15	58	0	15	58	0	17	68	0
27	63	23	128	2	23	128	2	27	151	0
42	104	38	239	2	38	239	2	110	780	1
77	191	71	845	6	71	845	6	(lookahead looping behaviour)		
115	291	120	1931	13	146	2297	7			
194	510	214	7522	22	359	12051	19			
677	2075	1011	126322	46	3600	>250000				

number of entries in the parsing tables are compared for non-probabilistic and probabilistic LALR parser generators, the latter incorporating the multiple-shift procedure discussed in section 2.1. Shift-reduce optimisation was applied in all cases. The "size" of each grammar is the total length of right-hand sides of all rules plus the number of nonterminal symbols. The number of table entries is the total of all non-error *action* and *goto* entries including multiple entries. Also displayed is the percentage (%>1) of non-error cells in the tables (*action* and *goto*) which contain multiple entries.

Only limited results are available for the canonical parser generator because the lookahead loop suppression procedure (section 2.2) has not yet been implemented. Despite the use of the canonical merging procedure (section 2.3) the size of the parsing tables is clearly growing rapidly and this version of parser generator is only a practical proposition for rather small grammars.

What stands out most from the LALR results is not that the total number of entries grows with the size of the grammar but that it does so exponentially. The space requirements of LR parsers for unambiguous computer languages tend to grow in a linear way with size (Purdom, 1974). It is also notable (and no coincidence) that the proportion of multiple entries also grows with the size of the grammar. Although further stages of optimisation may enable space to be saved, attention must be focussed on the grammar itself.

The parser generation algorithm of Pager (1977) is similar to the LALR algorithm except that states are merged only when doing so results in no additional multiple entries. All such entries are therefore the result of non-determinism in the grammar (with the exception of loop-breaking multiple shifts as discussed in section 2). This algorithm has been implemented for probabilistic grammars, but for the test series the results are identical to those for the LALR generator. It follows that the growing proportion of multiple entries is the product not of state merger but of rich non-determinism in the grammar.

The last two rows in the table

correspond to the addition to the grammar of two large groups of rules, used twice and once respectively in the corpus. These infrequent rules appear to introduce a high degree of ambiguity, which also shows up during the state-generation procedure. Each state is first generated as a "kernel" of items, and the presence of more than one item within a kernel implies that there is a local ambiguity which is being carried forward in order that the state automaton is deterministic. For the non-probabilistic LALR parser generator with the full grammar of 677 rules, the average kernel contained 6.1 items and the largest contained no fewer than 68!

According to Gazdar and Pullum (1985), it has never been argued that English is inherently ambiguous, rather that a descriptively adequate grammar should be ambiguous in order to account for semantic intuitions. However, the I.B.M. grammar may suffer from excessive ambiguity and the parser would benefit considerably if some way could be found to reduce it.

Finally, the physical storage requirements are easily stated: each table entry requires four bytes, two to specify the action and two for the probability in logarithmic form, converted to a short integer.

4. PROBABILISTIC PARSE FOREST

In keeping with the first and fourth objectives set out in the Introduction a probabilistic version of the parse forest representation of Tomita (1986) has been developed. In the presence of ambiguity, and even more so with uncertainty in the data, the number of interpretations may increase exponentially with the length of the input string. The impact of this is minimised by sub-node sharing and local ambiguity packing. Where two or more parses contain parts of their interpretation of a sentence which are identical they can share the relevant nodes. And, two or more parses may differ because of ambiguity which is localised: if part of the sentence is derivable from a nonterminal symbol in more than one way then the relevant nodes may be packed together. By thus compacting the parse forest the space requirement becomes $O(n^2)$ for most grammars (Kipps, 1989).

Employing this representation for probabilistic grammars requires that a value be attached to each node which enables the eventual calculation of the parse probability for the whole sentence given the data. In addition it is necessary that the m (say) most probable interpretations be obtained without an exhaustive search of the compacted parse forest.

A value $P(\Delta, \{D\}_{i\ldots j} \mid A)$ is attached to each node in a parse tree, where Δ denotes a particular derivation of the string $w_i \ldots w_j$ from the symbol A, and $\{D\}_{i\ldots j}$ represents the corresponding acoustical data. This probability is the product of the probabilities of all rules used in the particular derivation of $w_i \ldots w_j$ from A and the likelihoods of those words given the data, and is easily found for a particular node from the probabilities attached to each subnode and the rule probability when the reduce action occurs. This calculation is not affected by the context of $w_i \ldots w_j$, and therefore shared nodes need have only one value. For locally ambiguous packed nodes the probabilities of each alternative are recorded, in order that the correct ordering of alternatives can be created at further packed nodes higher in the

parse forest.

The probability attached to the S-node at the apex of any parse tree is $P(\Delta, \{D\}_{1\ldots M} \mid S)$ where M is the length of input. If all alternatives are retained in the parse forest then the parse probabilities given all the data can be obtained by normalisation. If all that is required is to identify the single most probable parse then all local ambiguity can be resolved by maximising at packed nodes (in the manner of the Viterbi algorithm) and retaining only the most probable derivation, because this ambiguity is invisible to higher-level structures in the forest.

The general problem of identifying the m most probable parses is more complex. The current m most probable are stored at each shared node together with a compact way of indicating which combination of subnodes corresponds to each derivation. Upon reduction by a rule whose right-hand side is of length k, the new m most probable derivations must be found and sorted from the (in the worst case) m^k possibilities. If two reductions are possible, to the same nonterminal symbol and spanning the same data, and if the right-hand sides are of length k_1 and k_2, then the worst-case number of possibilities is $m^{k_1} + m^{k_2}$, and so on. With appropriate book-keeping there are efficient ways to find and sort the m most probable of these, and record the subnodes. In practice this requires an array of size $4m$ stored for each node in the parse forest.

This approach has several advantages as compared with the original Bayesian algorithm for uncertain input data (Wright *et al*, 1989, 1990). The results are essentially equivalent, and most of the exponentially-growing number of possible interpretations are truncated away on grounds of probability, so the algorithm requires polynomial time and space. Furthermore, in this version the probabilities in the parsing tables are used only for prediction (to guide the pattern-matcher) and not for calculating the parse probabilities. These predictions do not have to be very precise so space can be saved by storing each probability in logarithmic form as a short integer. All this applies to isolated-word recognition; for connected speech the situation could be different.

5. ADAPTABILITY

The speed and effectiveness of the probabilistic LR parser would be seriously compromised if a large grammar (of, say, thousands of rules with a high degree of ambiguity) were adopted, and yet it would seem that if a grammar-based language model is to be employed for large-vocabulary speech recognition then the need for a large grammar will be unavoidable. The full version of the I.B.M. grammar referred to in section 3 extends to many thousands of rules but the greater part of these consist of oddball-rules that are used only once or twice in the corpus. Collectively the oddballs are important because they allow the corpus to be modelled, but individually each one is rather insignificant. It may be the case that generalisations (perhaps going beyond a context-free grammar) would eliminate a lot of these rules, but there is also a case for the parser to be made adaptive.

One approach to adaptation would be to assume a probabilistic grammar in Chomsky normal form and then use the inside-outside algorithm (Lari and Young, 1990). This approach has a lot to recommend it, but here we consider an alternative approach

based on a rule-adaptive enhancement of the LR parser. The principle is that at any time the parsing tables are based on a relatively small core grammar of important rules, but with an error-recovery procedure and a backup grammar. Error-recovery allows new rules to be created as required, and rules can be transferred between backup and core grammars in response to usage.

The probabilistic LR parser has been enhanced with such a procedure. The minimum adaptation which can allow an ungrammatical sentence to be accepted is a local change to a single existing rule: in this sense it is assumed that the sentence is "close" to the language. The conditions for rule-adaptation can be summarised as follows:

Input string:

$$w_1 \ldots . w_i \ldots . w_j \ldots . w_n$$

Existing rule: $A \rightarrow \alpha_1 \ \beta \ \alpha_2$

Adapted rule: $A \rightarrow \bar{\alpha}_1 \ \delta \ \bar{\alpha}_2$

such that

$$S \overset{*}{\Rightarrow} \gamma_1 \ A \ \gamma_2$$

$$\gamma_1 \ \bar{\alpha}_1 \overset{*}{\Rightarrow} w_1 \ldots . w_i$$

$$\delta \overset{*}{\Rightarrow} w_{i+1} \ldots . w_j$$

$$\bar{\alpha}_2 \ \gamma_2 \overset{*}{\Rightarrow} w_{j+1} \ldots . w_n$$

and where $\bar{\alpha}_1$, $\bar{\alpha}_2$ consist of α_1, α_2 with any unused nullable symbols suppressed.

The adaptation therefore consists in the deletion, insertion or replacement of a substring within the right-hand side of a rule, and the suppression of unused nullables simply ensures that all remaining symbols actually contribute to the parse of the sentence. This procedure usually generates a number of rule-candidates. Assuming that one of these is chosen as correct (although it may not be possible to automate this entirely), it is then added to the backup grammar as a potential core rule. With sufficient evidence of usage a rule may be promoted from backup to core grammar, and likewise a rule may be demoted.

A version is being developed in which the LR parsing tables are updated incrementally as rules are transferred between backup and core grammars. This should occur on-line, with the intention that the core grammar be kept reasonably compact (and the parser correspondingly fast) while adapting to the user. This is still very far from a complete solution to the problem of context-free grammar adaptation, but a system operating along these lines would satisfy (at least to some degree) all the objectives as set out in the Introduction.

ACKNOWLEDGMENT

This work was funded by the I.B.M. United Kingdom Scientific Centre.

REFERENCES

A V Aho, R Sethi and J D Ullman (1985), *Compilers: Principles, Techniques and Tools*, Addison-Wesley.

Gerald Gazdar and Geoffrey K. Pullum (1985), "Computationally relevant properties of natural languages and their grammars", *New Generation Computing*, 3, 273-306.

Fred Jelinek (1990), "Computation of the probability of initial substring generation by stochastic context free grammars", I.B.M. Research, Yorktown Heights, New York.

J R Kipps (1989), "Analysis of Tomita's algorithm for general context-free parsing", *Proceedings*

of the International Workshop on Parsing Technologies, Carnegie-Mellon University, 193-202.

K Lari and S J Young (1990), "The estimation of stochastic context-free grammars using the inside-outside algorithm", *Computer Speech and Language*, **4**, 35-56.

Chin-Hui Lee and Lawrence R. Rabiner (1989), "A frame-synchronous network search algorithm for connected word recognition", *IEEE Trans. on Acoustics, Speech and Signal Processing*, **37**, 1649-1658.

Kai-Fu Lee (1989), *Automatic Speech Recognition*, Kluwer Academic Publishers.

D Pager (1977), "A practical general method for constructing LR(k) parsers", *Acta Informatica*, **7**, 249-268.

P Purdom (1974), "The size of LALR(1) parsers", *BIT*, **14**, 326-337.

Richard Sharman (1989), "Observational evidence for a statistical model of language", I.B.M. United Kingdom Scientific Centre Report 205.

Masaru Tomita (1986), *Efficient Parsing for Natural Language*, Kluwer Academic Publishers.

Jerry Wright and Ave Wrigley (1989), "Probabilistic LR parsing for speech recognition", *Proceedings of the International Workshop on Parsing Technologies*, Carnegie-Mellon University, 105-114.

Jerry Wright (1990), "LR parsing of probabilistic grammars with input uncertainty for speech recognition", *Computer Speech and Language*, **4**, 297-323.

PHONOLOGICAL ANALYSIS AND OPAQUE RULE ORDERS

Michael Maxwell
Summer Institute of Linguistics
Box 248
Waxhaw, NC 28173 USA

ABSTRACT

General morphological/ phonological analysis using ordered phonological rules has appeared to be computationally expensive, because ambiguities in feature values arising when phonological rules are "un-applied" multiply with additional rules. But in fact those ambiguities can be largely ignored until lexical lookup, since the underlying values of altered features are needed only in the case of rare opaque rule orderings, and not always then.

INTRODUCTION

While syntactic parsing has a long and illustrious history, comparatively little work has been done on general morphological and phonological parsing — what I will call, for lack of a better term, "morphing."

The morphological and phonological parsing programs which do exist are, for the most part, either restricted to a single language or, like FONOL (Brandon 1988), are limited to generating surface forms from underlying forms. Two exceptions to this generalization are Kimmo (see Koskenniemi 1984, and the papers in *Texas Linguistic Forum* 22) and AMPLE (Weber, Black and McConnel 1988). However, Kimmo implements a non-standard theory of phonology, while AMPLE implements an item-and-arrangement morpher with virtually no allowance for (morpho-)phonological rules.

One reason for the paucity of general morphing programs is the apparent computational complexity of morphing. Phonological rules of natural language include deletion rules, which means that they potentially represent an unrestricted rewriting system. But in fact people routinely parse words into their constituent morphemes, which implies that Universal Grammar must place strong restrictions on phonology and morphology, effectively reducing the complexity of morphing. To the extent that linguists can analyze such restrictions, we may be able to reduce the computational complexity of morphing.[1] This paper investigates how one such restriction, a restriction on interaction among multiple rules, can be taken advantage of.

While linguists treat phonological rules as rules which derive surface forms from underlying forms, a program analyzing the surface strings of a language must "un-apply" those rules to a surface form to discover its underlying form. Most phonological rules have a neutralizing effect when applied in the derivational (synthesis) direction; accordingly, when a rule is un-applied, there will in general be more than one way to undo its effects. In a computational setting, this implies the need to restrict the search space, lest those ambiguities multiply with the application of multiple rules. This paper discusses a way of restricting that search space.

ASSUMPTIONS

For purposes of discussion, I will consider a morpher which implements a morphophonological theory of the following type. Phonological rules are written in the "standard" way with distinctive features but without any abbreviatory conventions (parentheses, curly braces, angled brackets, alpha variables, etc.); the rules apply in linear order, the output of each serving as the input to the next. I will assume that distinctive features are binary, although the results will apply in an analogous way to (finitely) multiply-valued features. For the most part, I will ignore the multiple application problem. I will not explicitly discuss morphological rules, but we may assume they apply either in a block (pre-cyclically) or cyclically. The resulting system resembles that of *The Sound Pattern of English* (Chomsky and Halle 1968, henceforth *SPE),* but without the abbreviatory schemata.

---

[1]Even without being able to explicitly state the restrictions, it may be that a correctly formulated set of rules for a given language will turn out to be readily parsable. However, that hope relies on the linguist to properly formulate the rules. I will return to this point later.

From a computational perspective, the working cycle of the morpher is as follows: phonological rules are un-applied in linear order to a form (assumed to be in an unambiguous phonetic representation), and then one or more morphological rules are un-applied (one in the case of cyclic rule ordering, one or more with non-cyclic rule ordering). By un-application of a rule, I mean applying it in reverse: going from a (more) surface form to a (more) underlying form. Lexical lookup is attempted after each morphological rule is un-applied. Lexical lookup acts then as a filter; if lexical lookup is successful, the set of phonological and morphological rules which were un-applied represents a successful derivation (modulo certain later tests, not discussed here), otherwise not. This is the classical approach to computational morphology/ phonology, as described in Kay (1977).[2]

THE PROBLEM

The problem to be explored in this paper arises when un-application of a rule results in one or more ambiguous feature values. Consider, as a simple case, the following rule:

$$\begin{bmatrix} +\text{syllabic} \\ -\text{cons} \\ -\text{stress} \end{bmatrix} \longrightarrow \emptyset \,/\, \begin{bmatrix} C \\ -\text{vd} \end{bmatrix} \text{——} \begin{bmatrix} C \\ -\text{vd} \end{bmatrix}$$

Suppose this rule is used to analyze a word which, on the surface, has two adjacent voiceless consonants. The rule specifies only three features for the vowel to be epenthesized in analysis of the surface form (i.e. the vowel which was deleted to generate the surface form): [+syllabic -cons -stress]. The remaining features must be "guessed" during analysis. Since this involves multiple features, the combinatorial possibilities are many. In addition, there is the possibility that no vowel should be epenthesized — that the consonants were adjacent underlyingly.

---

[2]The morpher discussed in the text is being implemented as one module of the planned "Hermit Crab" system (a syntactic parser and possibly a functional structure module being additional modules). Hermit Crab takes its name from the fact that the internal rule system (the "crab") has a rule structure which will, in general, depart from the rule structure as viewed by the user (who sees only the "shell").

This problem is not limited to rules of deletion. Any rule which neutralizes an underlying contrast will cause ambiguity (albeit not usually as great as in the case of deletion rules) when the rule is un-applied.

The difficulty is compounded by the interaction of multiple rules. Anderson (1988) suggests a "typical" rule depth in natural languages of 15-20 rules. Clearly the possibilities of computational explosion loom large.

The remainder of this paper will investigate some approaches to this problem.

THE BRUTE FORCE APPROACH

I will first explore the following brute-force technique: when a phonological rule is un-applied, instantiate all possible combinations of features changed by the rule onto the new form output by the rule.

For a deletion rule, the number of feature combinations which may be instantiated is 2^n, where n = the number of features not specified in the left-hand side of the rule. For concreteness, consider the vowel deletion rule discussed above. In the SPE system I count eighteen distinctive features (not including certain prosodic features). Subtracting the three features whose values are supplied by the rule leaves fifteen unspecified features. Since 2^{15} is a very large number, there is clearly a need for pruning the search space.

A certain amount of pruning comes readily. One can begin by eliminating universally impossible feature cooccurences. For instance, if a segment is [+syllabic], it cannot be [-continuant]. In the case of the vowel deletion rule, this reduces the search space to about 2^8 combinations. (The eight features in the SPE system whose values are not determined by the [+syll -cons -stress] features of the rule are: High, Low, Back, Round, Tense, Voiced, Covered and Nasal. Some combinations of these are also mutually incompatible, e.g. [+high +low], reducing the search space slightly more.) We can do still better by eliminating noncontrastive features in the language we are working with. For Spanish, for instance, we could eliminate the features *Covered* and *Nasal* if we work with the surface vowels (ignoring the light nasalization of vowels before nasal consonants), and the features *Tense* and *Voiced* if we limit ourselves to features appearing only in

underlying vowels. (The assumption here is that tensing and voicing, which in most dialects of Spanish are predictable, do not condition other rules.) These reductions leave a search space of $2^4 = 16$. We can limit this still further by eliminating combinations of features which do not occur in a particular language ([-back +round], for instance). We are left with an irreducible search space of five combinations of features in this case — the five vowels which occur (underlyingly) in standard Spanish. This last reduction constitutes the use of Segment Structure Conditions (SSCs) to constrain rule un-application. This may be done rapidly by consulting a list of possible segments of the language. (The list of possible segments need not be confined to those appearing at the surface; i.e. absolute neutralization can be accommodated by allowing for absolutely neutralized segments in the SSCs.)

Since this rule is a deletion rule, we must also allow for the situation in which no vowel was deleted, increasing the search space by one.

Similarly, non-deletion rules introduce an ambiguity of 2^m, where m = the number of features on the right-hand side of the rule, often pruneable by reference to SSCs.[3] Consider, for example, a language in which the only coronal obstruants are *t* and *č*, and the following rule:

$$\begin{bmatrix} +\text{cor} \\ -\text{cont} \end{bmatrix} \longrightarrow \begin{bmatrix} -\text{ant} \\ +\text{del rel} \end{bmatrix} \; / \; \_\_ \text{i}$$

Naive un-application of this rule to the sequence *či* would lead to a four-way ambiguity in the values of the feature set {ant, del rel}; but this ambiguity can be reduced to a two-way ambiguity by the use of SSCs in combination with the known features on the left-hand side of the rule, since two combinations ([+ant +del rel] and [-ant -del rel]) can be ruled out. In general, the more features there on the right hand side of the rule (and hence the more ambiguous the underlying feature values, apart from pruning), the more likely it is that some combinations of those features can be ruled out. It is clear, then, that using SSCs considerably improves the Brute Force method.

Thus far, I have considered only the case where a single rule is un-applied, without regard for other rules, nor for the possible reapplication of the rule in question. The interaction of several rules results in a combinatorial multiplication: the number of feature values which must be instantiated in the course of analysis is (roughly) the product of the number of feature values which must be instantiated during the un-application of each rule. This combinatorial explosion is one of the major reasons it has seemed that the automatic un-application of phonological rules is a computationally difficult problem. This problem of multiple rule application will be the topic of the next section.

The effects of a rule which can re-apply to its own output can be even worse. Consider the following plausible consonant cluster simplification rule:

$$\text{C} \longrightarrow \emptyset \; / \; \text{C} \_\_\_\_ \text{C}$$

If this rule is un-applied to a surface form with a two-consonant cluster, the result will be an intermediate form having a three-consonant cluster. But if the rule is allowed to un-apply to this intermediate form, it can un-apply in two places to yield a five-consonant cluster, and so on ad infinitum! There are two ways of avoiding this problem: placing ad hoc limits on the application of deletion rules (which are the only rules that can cause such infinite application), or requiring that the forms derived by reverse application of phonological rules meet certain conditions, such as Morpheme Structure Conditions.

Morpheme Structure Conditions (MSCs) would be the most principled solution. Nonetheless, a

---

[3] An assumption here is that the features on the left- and right-hand sides of the rule are disjoint. Anyone who has taught phonology has seen students write rules like the following:

$$\begin{bmatrix} \text{C} \\ +\text{vd} \end{bmatrix} \longrightarrow [\text{-vd}]$$

/ (some environment)

The *+vd* specification on the left-hand side is redundant; without it, the rule applies vacuously to underlyingly nonvoiced consonants. The phonological literature as well contains many such rules with redundant specifications, but they can usually be reanalyzed to eliminate the redundancy. Of the few rules which resist reanalysis, most employ such debatable techniques as alpha switching variables or angled brackets. I leave it to phonologists to determine whether rules which necessarily employ the same features on both sides of the arrow actually occur in natural languages.

morphing program must rely on the linguist to write rules and conditions which in their combination will not cause problems. Nor are such interactions always obvious. For instance, the above rule could be written to delete consonants only at morpheme boundaries:

C ⟶ Ø / C ___ +C

Then if morphemes of a single consonant are allowed, MSCs would not prevent the rule from looping infinitely, endlessly postulating deleted morphemes. (I assume here that morpheme boundaries, unlike other parts of the environment, must be postulated as needed during un-application of phonological rules, since they are unlikely to be marked in surface forms. Clearly such postulation will have to be restricted. See Barton, Berwick and Ristad 1987, sec. 5.7, concerning problems caused by unrestricted postulation of segments which are null at the surface.) Furthermore, it has often been proposed that MSCs do not apply to the output of phonological rules (Kenstowicz and Kisseberth 1977, chap. 3, and Anderson 1974, chap. 15). Hence, ad hoc limits on rule reapplication will be needed, even with MSCs.

One might hope that Word Structure Conditions (WSCs) would have the desired effect in constraining rules. However, since WSCs apply to surface forms, they cannot help. In fact, from one perspective a consonant deletion rule exists in order to bring a nonconforming underlying representation into conformance with a WSC; hence the un-application of such a rule necessarily results in an intermediate form violating the WSC.

WHEN MUST FEATURES BE INSTANTIATED?

In this section I explore an approach to the problem of multiple rule interaction and the resulting combinatorial explosion. I will argue that features altered by rules can usually be left un-instantiated (at least until lexical lookup), thereby avoiding the combinatorial effects otherwise inherent in multiple rule application. The question then is, Under what circumstances do features actually need to be instantiated during un-application of a rule?

As a first approximation, if one rule assigns a value to some feature, while the environment of an earlier rule refers to that same feature, it may be necessary to instantiate the feature values altered by the later rule. I will refer to the situation where such instantiation becomes necessary as "interference" between the two rules.

We can be more precise about when interference occurs, since two rules do not interfere if the second rule can only alter the feature in an environment in which the first rule could not apply.[4]

Before giving a more explicit definition of when two rules interfere, I present a definition of *phonological unification:*

> Let $X = X_1 \ldots X_a \ldots X_i$ and $Y = Y_1 \ldots Y_b \ldots Y_j$ be two non-empty phonological sequences (i.e. sequences of segments, each segment being a set of distinctive features). Then X and Y *p[honologically]-unify,* with X_a and Y_b *corresponding,* iff there exists a phonological sequence $Z = Z_1 \ldots Z_g \ldots Z_k$ such that Z_g is the unification of X_a and Y_b, and all the other segments of Z are the unifications of the respective segments of X and Y (where X and Y may be extended to the left and/or right as necessary by the addition of empty segments).

Consider then the following two rules:[5]

A ⟶ B / C _ D

E ⟶ F / G _ H

---

[4]A concrete example of a situation where there is no interference, despite the fact that the second rule alters a feature referred to by the first rule, is the following two hypothetical rules:

$$\begin{bmatrix} C \\ -vd \end{bmatrix} \longrightarrow [+asp] \,/\, \_\_\_\ V$$

C ⟶ [-vd] / ___ #

Since the second rule devoices consonants only word finally, it will never interfere with the first rule, which refers to voiceless consonants only in pre-vocalic position. I asume here that there is no rule of word-final vowel deletion ordered between these rules; see fn. 6.

[5]I assume the features on the two sides of each rule are disjoint; see fn. 3.

In order to un-apply the first rule to a form, the values of the features given in A, B, C, and D must be known in that form, so that they can be matched against the values required by the rule. Interference occurs when the second rule alters any of the features of A, B, C, or D in an environment compatible with the application of the first rule. More specifically, let W (the output of the first rule) = C (A ∪ B) D, where (A ∪ B) = the unification of A and B; and let w_i be a segment of W such that w_i includes one or more of the features of F (i.e. the features altered by the second rule, not necessarily with the values specified in F). Then the second rule will interfere with the first if G E H p-unifies with W such that the segment E corresponds to w_i.

If we restrict our attention to the case where the output of the second rule contains but a single feature, the interference just described corresponds to one of two types of rule interactions in phonological theory: counterbleeding and counterfeeding interactions. To see why, suppose the order of application of the two rules were reversed. Then the rule E ⟶ F (now the first rule to apply) assigns certain values to the feature F, while the other rule relies on a certain value of that feature being present in its environment. Furthermore, the two environments are compatible (p-unifiable), by hypothesis. Then if the first rule (E ⟶ F) assigns the required value to F, it feeds the second rule (A ⟶ B). Similarly, if the first rule assigns to F the opposite of the required value, it bleeds the second rule. Since the actual rule order is the reverse, the rules stand in either a counterfeeding or a counterbleeding relationship.

The reason for restricting attention to one feature of F at a time, is that the rules may stand in a counterfeeding relationship with respect to one feature, but a counterbleeding relationship with respect to another feature. In such a case, the pair of rules as a whole will be in a counterbleeding relationship. (There may also be features in F which do not cause interference.)

It is significant that counterfeeding and counterbleeding rule orders are precisely those rule orders which are opaque (cf. Kiparsky 1971). In other words, the features altered by the second rule will have to be instantiated just in case the two rules are opaquely ordered.[6] Crucially, opaque rule orderings appear to be quite rare in natural language (Kiparsky 1971). (More precisely, rules which are opaquely ordered tend to be lost, reordered, or reanalyzed, so that opaque orderings are unstable. As a result, they tend to be rare.)

The fact that interference only occurs with opaque rule orderings suggests a better method of rule un-application than the Brute Force approach: instantiate features in a rule only if they (potentially)

---

[6]This description of potentially interfering rules is complicated by the fact that a rule ordered between two other rules can change their interaction. Consider the following two rules:

$$k \longrightarrow k^h \,/\, \_\_\_ \begin{bmatrix} V \\ +high \\ +stress \end{bmatrix}$$

$$\begin{bmatrix} V \\ -stress \end{bmatrix} \longrightarrow [-high] \,/\, \_\_\_ \begin{bmatrix} V \\ -high \end{bmatrix}$$

Since these rules are not p-unifiable (the [-stress] feature requirement in the second rule not being unifiable with the [+stress] feature in the first rule), the rules cannot interfere as they stand. But now let a second rule be introduced, ordered between these two, so that the new set of rules is the following:

$$k \longrightarrow k^h \,/\, \_\_\_ \begin{bmatrix} V \\ +high \\ +stress \end{bmatrix}$$

$$V \longrightarrow [-stress] \,/\, \_\_\_ \; C\,V\,C \begin{bmatrix} V \\ +stress \end{bmatrix}$$

$$\begin{bmatrix} V \\ -stress \end{bmatrix} \longrightarrow [-high] \,/\, \_\_\_ \; C \begin{bmatrix} V \\ -high \end{bmatrix}$$

The first rule is now (potentially) interfered with by both of the other rules: the second rule alters the stress on the vowel, and the third rule alters the height of that vowel in an environment which may now be compatible with that of the first rule because of the destressing rule.

For an intermediate rule to alter the interaction of two other rules, the intermediate rule must be p-unifiable with both the other rules (otherwise it could not operate on any forms that both the other rules operated on); and it must change the value of the feature(s) on the first rule which block p-unifiability with the third rule into a value(s) compatible with the third rule, i.e. feed the third rule.

interfere with an earlier rule. Instead, when a rule is un-applied, simply mark the features it changes as uninstantiated (i.e. of unknown value). In practice, this will usually result in a large savings in search space due to the rarity of opaque rule orderings. It will still be necessary to instantiate these "empty" features prior to lexical lookup, but this is clearly a lesser problem, since the effects are not multiplicative (and the instantiation can again be restricted by reference to SSCs).

However, in the next section I will suggest an even better approach, which takes advantage of the fact that even though two rules interfere potentially, the interference may not arise in every word in which one or the other of the rules applies. (In fact, one can imagine that in a language having potentially counterbleeding or counterfeeding rules, it might be the case that no words actually meet the structural description of both rules.)

THE LAZY APPROACH: INSTANTIATING FEATURES ONLY WHEN NECESSARY

The strategy of the lazy approach should by now be clear: postpone instantiation of feature values altered by phonological rules until those values are actually needed, either by lexical lookup or in order to un-apply another rule (i.e. when all the instantiated features of a form match a rule, but the values of one or more uninstantiated features in the form are also specified by the rule). When features are instantiated, such instantiation may again be restricted by the SSCs. In effect, then, un-application of a phonological rule produces archiphonemes,[7] so that features are instantiated only when absolutely required. Assuming that opaque rule orders are as rare as phonologists have claimed, and that words in which both members of a pair of opaque rules apply are even rarer, this will greatly reduce the computational complexity of general computational morphology.

[7]I assume that all features start out instantiated in surface forms, even "irrelevant" features. If they were not, it would be impossible on examining a given un-instantiated feature to know whether it has become un-instantiated during the course of the derivation and therefore is a candidate for instantiation, or whether it is an irrelevant feature for a particular segment and therefore could not trigger the rule in question. Alternatively, one could keep track of which features have become un-instantiated during the (un-)derivation.

CONCLUSIONS

In summary, I have shown that one of the combinatorial difficulties which would appear to make implementation of general morphing programs impractical is the ambiguity of feature values arising during un-application of phonological rules. But in fact those ambiguous values are needed later in the derivation only in the case of opaque rule orderings. This apparent difficulty can therefore be dealt with by delaying the instantiation of features which have become un-instantiated until they are actually required. Since opaque rule orderings are relatively rare, this results in a considerable savings in search space against the alternative of immediately instantiating all features altered by rules. Delayed instantiation also represents a savings in search space against the alternative of instantiating only those features whose values may be required by another rule, since not all words will meet the structural description of both rules of an opaque pair.

REFERENCES

Anderson, Stephen R. 1974. *The Organization of Phonology.* New York: Academic Press.

Anderson, Stephen R. 1988. "Morphology as a Parsing Problem." In *Morphology as a Computational Problem,* ed. Karen Wallace, 1-21. UCLA [Linguistics Department] Occasional Papers no. 7, Working Papers in Morphology.

Barton, G. Edward, Robert C. Berwick, and Eric Sven Ristad. 1987. *Computational Complexity and Natural Language.* Cambridge, Mass.: MIT Press.

Brandon, Frank R. 1988. "FONOL: Phonological Programming Language." Unpublished computer file.

Kay, Martin. 1977. "Morphological and Syntactic Analysis." In *Linguistic Structures Processing,* ed. Karen Wallace, 131-234. Amsterdam: North Holland.

Kenstowicz, Michael, and Charles Kisseberth. 1977. *Topics in Phonological Theory.* New York: Academic Press.

Kiparsky, Paul. 1968. "Linguistic universals and linguistic change." In *Universals in Linguistic Theory,* ed. Emmon Bach and Robert Harms, 171-202. New York: Holt, Rinehart and Winston.

Kiparsky, Paul. 1971. "Historical Linguistics." In *A Survey of Linguistic Science,* ed. William Dingwall, 576-642. College Park: University of Maryland.

Kisseberth, Charles. 1970. "On the Functional Unity of Phonological Rules." *Linguistic Inquiry* 1:291-306.

Koskenniemi, Kimmo. 1984. "A general computational model for word-form recognition and production." *COLING-84* 178-181.

Stanley, Richard. 1967. "Redundancy rules in phonology." *Language* 43:393-436.

Various, 1983. *Texas Linguistic Forum* no. 22. Department of Linguistics, University of Texas at Austin.

Weber, David J., H. Andrew Black, and Stephen R. McConnel. 1988. *AMPLE: A Tool for Exploring Morphology.* Occasional Publications in Academic Computing. Dallas: Summer Institute of Linguistics.

February 14, 1991

Session B

AN EFFICIENT CONNECTIONIST CONTEXT-FREE PARSER

Klaas Sikkel and Anton Nijholt
Department of Computer Science, University of Twente,
PO box 217, 7500 AE Enschede, The Netherlands
sikkel@cs.utwente.nl
anijholt@cs.utwente.nl

ABSTRACT

A connectionist network is defined that parses a grammar in Chomsky Normal Form in logarithmic time, based on a modification of Rytter's recognition algorithm. A similar parsing network can be defined for an arbitrary context-free grammar. Such networks can be integrated into a connectionist parsing environment for interactive distributed processing of syntactic, semantic and pragmatic information.

INTRODUCTION

Connectionist networks are strongly interconnected groups of very simple processing units. Such networks are studied in natural language processing since their inherent parallelism and distributed decision making allows an integration of syntactic, semantic and pragmatic processing for language analysis. See, e.g., (Waltz and Pollack, 1988), (Cotrell and Small, 1989). By isolating the syntactic component — without abandoning the connectionist paradigm — it becomes possible to study context-free parsing in environments where we can make different assumptions about types of networks, learning rules and representations of concepts. Examples of this type of research can be found in (Fanty, 1985), a simple connectionist implementation of the CYK method; (Selman and Hirst, 1987), Boltzmann machine parsing; (Howells, 1988), a relaxation algorithm that utilizes decay over time; (Nakagawa and Mori, 1988), a parallel left-corner parser incorporated in a learning network; (Nijholt, 1990), a Fanty-like connectionist Earley parser.

In this paper we push the speed of the parsing network to its limits, so as to investigate how much parallellism is possible in principle. We define a parsing network that constructs a shared forest of parse trees in $O(\log n)$ time for an input string of length n, using $O(n^6)$ units. Our network is based upon Fanty's "dynamic programming" approach and a type of algorithm first introduced by Rytter (1985). The network is rather large, but not too large: no logarithmic-time parsing algorithm for arbitrary contrext-free languages is known that uses less than $O(n^6)$ processors. Furthermore, the number of units can drastically be reduced (albeit within the same complexity bounds) by a meta-parsing algorithm that constructs a minimal network custom-tailored for a specific grammar.

After some preliminary definitions, we construct a network for a grammar in Chomsky Normal Form. At the end of this paper we argue that a similar network can be built for an arbitrary CFG; space limitations do not allow a detailed presentation.

PRELIMINARIES AND DEFINITIONS

Let $G = (N, \Sigma, P, S)$ be a grammar in Chomsky Normal Form (CNF), i.e., production rules have the form $A \rightarrow BC$ or $A \rightarrow a$. We consider input strings $a_1 \cdots a_m$ with $m < n$, where n is an implementation-dependent constant.

A central role in the parsing algorithm is played by items of the form $\langle A,i,j \rangle$, which are called *triangles*. A triangle $\langle A,i,j \rangle$ is called *recognizable* if $A \Rightarrow^+ a_{i+1} \cdots a_j$. The set of triangles Ξ is defined by

$$\Xi \stackrel{\text{def}}{=} \{\langle A,i,j \rangle \mid A \in N,\ 0 \le i < j \le n\}.$$

A triangle $\langle A,i,j \rangle$ is called *parsable* if it is recognizable and $S \Rightarrow^+ a_1 \cdots a_i A a_{j+1} \cdots a_m$. The collection of all parsable triangles is called the *shared forest* of an input sentence; "forest" because it comprises all different parse trees for that sentence, "shared" as common sub-trees of different parse trees are represented only once. The algorithm and network in section 3 compute the shared forest of a sentence.

We shall also need items of a different kind, called *triangles with a gap*, denoted $\langle\langle A,i,j \rangle, \langle B,k,l \rangle\rangle$. A triangle with a gap $\langle\langle A,i,j \rangle, \langle B,k,l \rangle\rangle$ is called *proposable* if $A \Rightarrow^+ a_{i+1} \cdots a_k B a_{l+1} \cdots a_j$. During the application of the algorithm, we will propose triangles with a gap that need further investigation. If $\langle\langle A,i,j \rangle, \langle B,k,l \rangle\rangle$ has been proposed and $\langle B,k,l \rangle$ can be recognized, we can fill up the gap and recognize $\langle A,i,j \rangle$. The set of all triangles with a gap is denoted by

$$\Gamma \stackrel{\text{def}}{=} \{\langle\langle A,i,j \rangle, \langle B,k,l \rangle\rangle \mid \langle A,i,j \rangle \in \Xi,\ \langle B,k,l \rangle \in \Xi,\ i \le k < l \le j, i \ne k \text{ or } l \ne n\}.$$

The gap can be at the inside or at the outside of a tri-

angle, as shown in Figure 1.

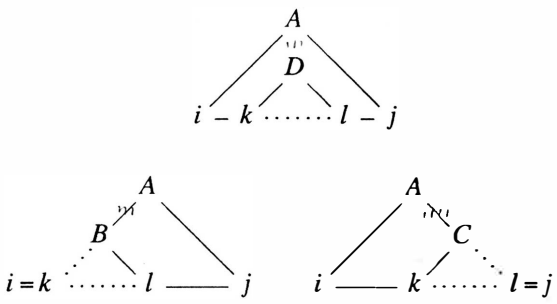


Figure 1. Triangles with a gap

The *size* of a triangle is defined as the length of the substring $a_{i+1} \cdots a_j$: $size(\langle A,i,j\rangle) = j-i$. The size of a triangle with a gap is defined as the size of the triangle minus the size of the gap: $size(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle) = size(\langle A,i,j\rangle) - size(\langle B,k,l\rangle) = j-i-l+k$.

A FAST CONNECTIONIST PARSING NETWORK FOR CNF GRAMMARS

A variant of Rytter's recognition algorithm

The algorithm presented here is a (for our purpose) improved version of Rytter's recognition algorithm (Gibbons and Rytter, 1988). It can be trivially extended into a parsing algorithm and has a simpler correctness proof. Remarks about the differences with the original algorithm are deferred to the end of this section, so as to keep the exposé as clear as possible. We will describe first what is to be computed by the algorithm, and elaborate on how to compute it afterwards. The recognition algorithm uses two tables of boolean values:

- $recognized(\langle A,i,j\rangle)$ for $\langle A,i,j\rangle \in \Xi$, which is *true* once we have established that $\langle A,i,j\rangle$ is indeed recognizable, and *false* otherwise;
- $proposed(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle)$ for $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle \in \Gamma$, which is *true* once we have established that $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle$ is indeed proposable, and *false* otherwise.

The algorithm will satisfy the following loop-invariant properties:

(I) if $size(\langle A,i,j\rangle) \le 2^k$ and $\langle A,i,j\rangle$ is recognizable then $recognized(\langle A,i,j\rangle) = true$ after k steps,

(II) if $size(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle) \le 2^k$ and $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle$ is proposable then $proposed(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle) = true$ after k steps.

Acceptance or rejection of the input string depends on the recognizability of $\langle S,0,m\rangle$, hence the number of steps that need to be performed is $\lceil {}^2\log m\rceil$ (the smallest integer $\ge {}^2\log m$).

The well-known Cocke-Younger-Kasami algorithm uses an upper triangular recognition table T_{CYK}: The nonterminal A is added to table entry $t_{i,j}$ if $\langle A,i,j\rangle$ is recognized. The statements $A \in t_{i,j}$ and $recognized(\langle A,i,j\rangle) = true$ are equivalent. We can illustrate the results of our algorithm (though the operations are different) with an extension of the T_{CYK} recognition table. In this case, the recognition table is a three-dimensional structure,

$$T_R = \{t_{i,j,k} \mid 0 \le i < j \le n,\ 0 \le k \le j-i\}.$$

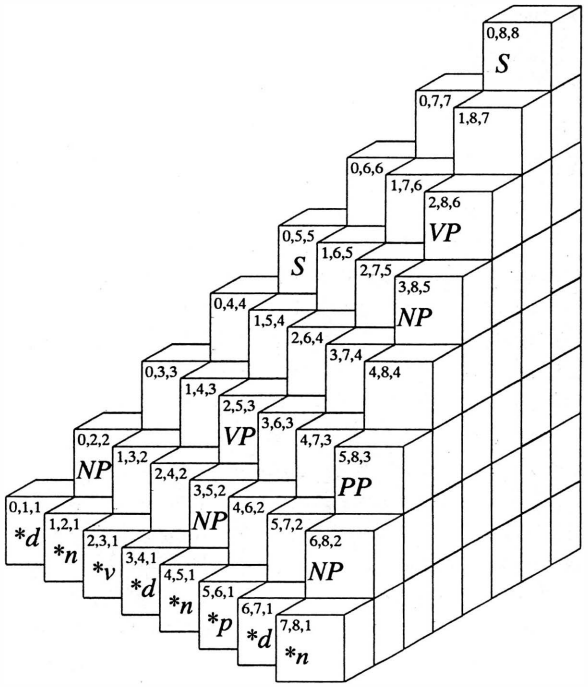


Figure 2. The surface of T_R as it should be computed by the algorithm

The third index k denotes the size of an item. When a triangle $\langle A,i,j\rangle$ is recognized, the nonterminal A is added to $t_{i,j,j-i}$. When a triangle with a gap $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle$ is proposed, an object $\langle A,B,k,l\rangle$ is added to $t_{i,j,h}$ with $h = j-i-l+k = size(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle)$. Hence the surface of T_R is equal to T_{CYK}, representations of triangles with a gap are contained in entries inside the table. Invariants (I) and (II) guarantee that a table entry with height k will be completed within $\lceil {}^2\log k\rceil$ steps. As a simple example, consider the grammar

$$S \rightarrow NP\ VP \mid S\ PP$$
$$NP \rightarrow *det\ *noun \mid NP\ PP$$
$$VP \rightarrow *verb\ NP$$
$$PP \rightarrow *prep\ NP$$

and the input sentence *the boy saw a man with a telescope*. Figure 2 shows the surface of T_R after application of the algorithm.

Having illustrated the purpose of the recognition algorithm, we can now explain how it works. We define the following operations on Ξ, Γ and the tables *recognized* and *proposed*;

INITIALIZE :
for all $\langle A,i,j\rangle \in \Xi$
do *recognized*$(\langle A,i,j\rangle)$:= *false* **od** ;
for all $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle \in \Gamma$
do *proposed*$(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle)$:= *false* **od** ;
for all $\langle A,i-1,i\rangle \in \Xi$
do **if** $A \rightarrow a_i \in P$
then *recognized*$(\langle A,i-1,i\rangle)$:= *true* **fi**
od

PROPOSE :
for all $\langle A,i,j\rangle, \langle B,i,k\rangle, \langle C,k,j\rangle \in \Xi$
such that $A \rightarrow BC \in P$
do **if** *recognized*$(\langle B,i,k\rangle)$
then *proposed*$(\langle\langle A,i,j\rangle,\langle C,k,j\rangle\rangle)$:= *true* **fi** ;
if *recognized*$(\langle C,k,j\rangle)$
then *proposed*$(\langle\langle A,i,j\rangle,\langle B,i,k\rangle\rangle)$:= *true* **fi**
od

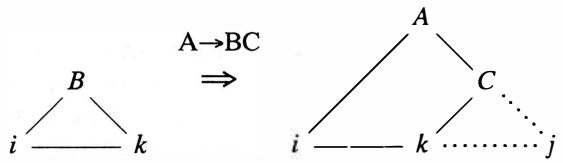


Figure 3. *PROPOSE*

RECOGNIZE :
for all $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle \in \Gamma$, $\langle B,k,l\rangle \in \Xi$
do **if** *proposed*$(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle)$
and *recognized*$(\langle B,k,l\rangle)$
then *recognized*$(\langle A,i,j\rangle)$:= *true* **fi**
od

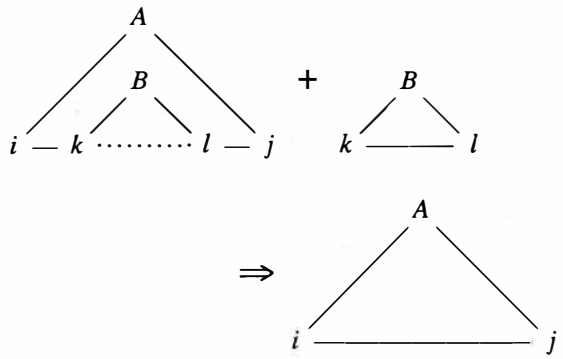


Figure 4. *RECOGNIZE*

COMBINE :
for all $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle$, $\langle\langle B,k,l\rangle,\langle C,m,n\rangle\rangle \in \Gamma$
do **if** *proposed*$(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle)$
and *proposed*$(\langle\langle B,k,l\rangle,\langle C,m,n\rangle\rangle)$
then *proposed*$(\langle\langle A,i,j\rangle,\langle C,m,n\rangle\rangle)$:= *true* **fi**
od

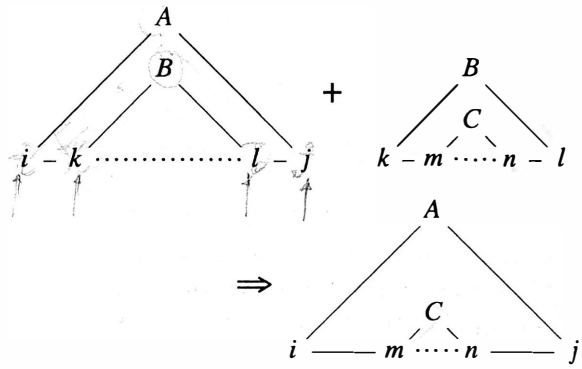


Figure 5. *COMBINE*

The functioning of the operators *PROPOSE*, *RECOGNIZE* and *COMBINE* is illustrated in Figures 3–5. Everything in a **for all** statement can be computed in parallel. The recognition algorithm, using these operators, can be given as:

begin
INITIALIZE ;
PROPOSE ;
repeat $\lceil {}^2\log m \rceil$ **times**
begin
RECOGNIZE ;
PROPOSE ;
COMBINE ;
COMBINE
end ;
if *recognized*$(\langle S,0,m\rangle)$
then *accept*
else *reject*
fi
end

In the sequel, we will give a proof of the correctness of the modified Rytter algorithm. But let's first look at an example.

In Figure 6 one parse tree of the input sentence is shown. The algorithm obviously recognizes much more than a single parse tree, but it is sufficient to show that all items in one parse tree are recognized in order to make clear that the top item is recognized. $\langle S,0,8\rangle$ can be recognized in a number of different ways, but that would only clutter up the example. The nodes in the parse tree have been numbered, so

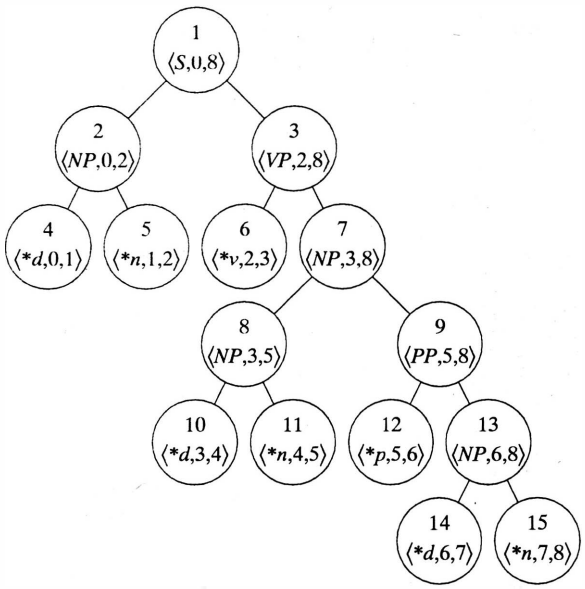


Figure 6. A parse tree

We can identify the triangles by their number rather than by the more cumbersome $\langle A, i, j \rangle$ notation. We will apply the algorithm step-by-step on the items in this tree. Step 0 is shown in Figures 7–8, step 1 in Figures 9–12 and (the first half of) step 2 in Figures 13–14. Circles correspond to recognized triangles, lines correspond to proposed triangles with gaps. The example shows that the algorithm may need less than $\lceil {}^2\log k \rceil$ steps in some cases; we need only 2 steps although 3 are allowed.

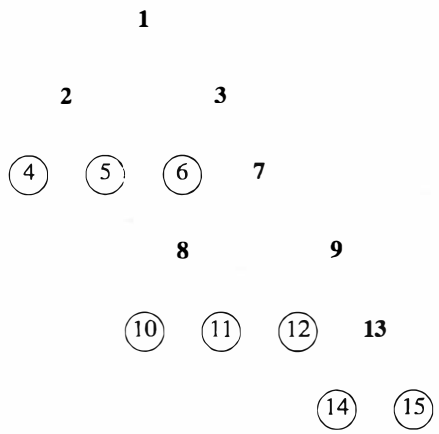


Figure 7. After step 0(a): *INITIALIZE*

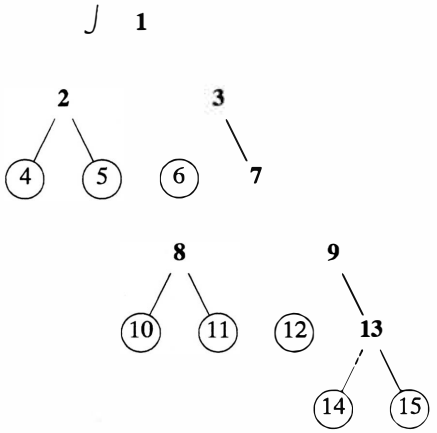


Figure 8. After step 0(b): *PROPOSE*

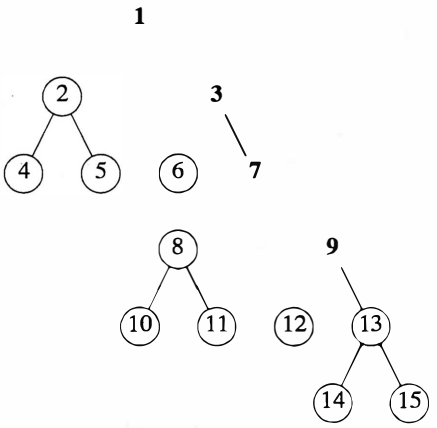


Figure 9. After step 1(a): *RECOGNIZE*

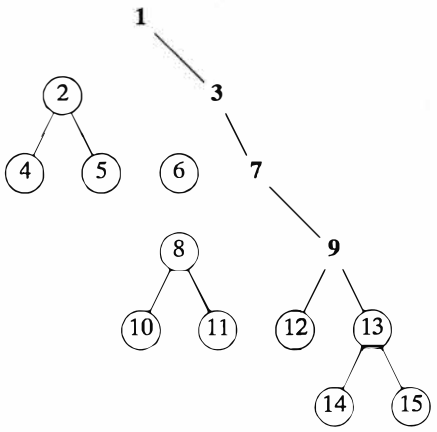


Figure 10. After step 1(b): *PROPOSE*

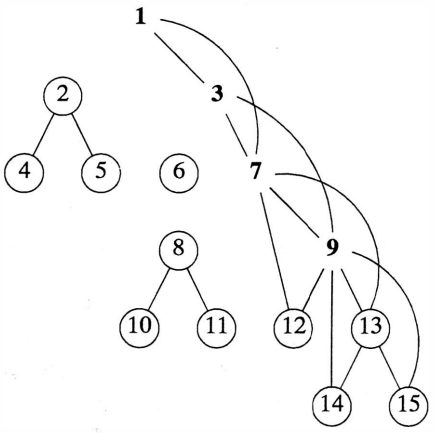


Figure 11. After step 1(c): *COMBINE*

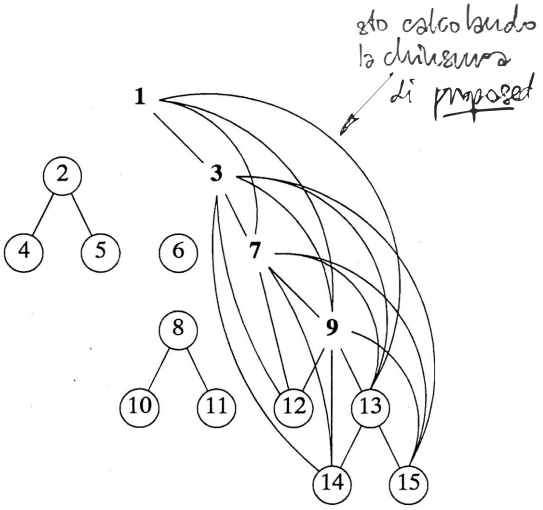


Figure 12. After step 1(d): *COMBINE*

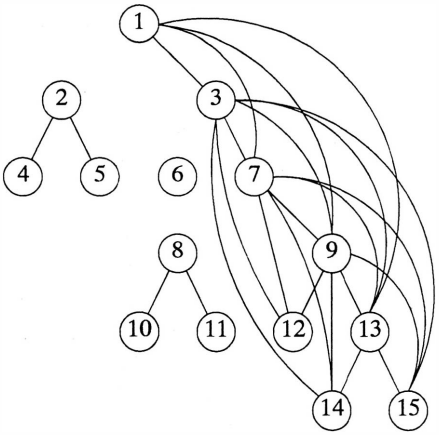


Figure 13. After step 2(a): *RECOGNIZE*

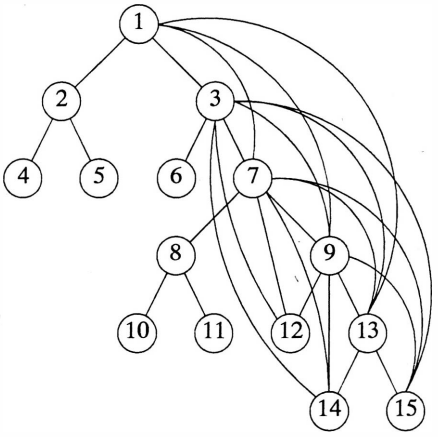


Figure 14. After step 2(b): *PROPOSE*

Correctness of the algorithm

Theorem. *After application of the above algorithm, a triangle will have been recognized if and only if it is recognizable; a triangle with a gap will have been proposed if and only if it is proposable.*

We will give a proof that is a great deal simpler than the proof of the original algorithm (Gibbons and Rytter, 1988). For more details see (Sikkel and Nijholt, 1990).

Terminology. We denote triangles with greek letters ξ, η, ζ etc. The triangles η, ζ are called *a pair of sons of* ξ if $\xi = \langle A,i,j\rangle$, $\eta = \langle B,i,k\rangle$, $\zeta = \langle C,k,j\rangle$ for some $A, B, C \in N$ with $A \rightarrow BC \in G$ and $0 \le i < k < j \le n$. For technical reasons we allow empty triangles with a gap $\langle \xi, \xi\rangle$. For such an empty triangle, *proposed*$(\langle \xi, \xi\rangle)$ = *true* by definition.

Basis. It is easy to verify that proposable triangles with a gap of size 1 have been proposed after the initialization step and recognizable triangles of size ≤ 2 have been recognized after step 1.

Induction hypothesis. We write $(\mathrm{I})_k$ for the claim that (I) holds for ξ with $size(\xi) \le 2^k$ and $(\mathrm{II})_k$ for the claim that (II) holds for $\langle \xi, \eta\rangle$ with $size(\langle \xi, \eta\rangle) \le 2^k$. Hence $(\mathrm{II})_0$ and $(\mathrm{I})_1$ have been established above. From the induction hypothesis $(\mathrm{II})_{k-1}$, $(\mathrm{I})_k$ we will derive $(\mathrm{II})_k$, $(\mathrm{I})_{k+1}$.

(II)$_k$. Given $(\mathrm{II})_{k-1}$, $(\mathrm{I})_k$, we prove $(\mathrm{II})_k$. Let $\langle \xi, \eta\rangle$ be proposable, $2^{k-1} < size(\langle \xi, \eta\rangle) \le 2^k$.

- **Claim A**. There is a ϕ with sons ψ, ψ', such that ϕ, ψ, ψ' are recognizable, $\langle \xi, \phi\rangle$, $\langle \psi, \eta\rangle$ are proposable, $size(\langle \xi, \phi\rangle) \le 2^{k-1}$, $size(\langle \psi, \eta\rangle) \le 2^{k-1}$. See Figure 15.

  Proof. If $\langle \xi, \eta\rangle$ is proposable, there is a sequence $\zeta_0, \cdots, \zeta_p$ with $\zeta_0 = \xi$, $\zeta_p = \eta$ such that each

$\langle \zeta_i, \zeta_{i+1} \rangle$ is proposed by a *PROPOSE* operation; these "atomic" triangles with a gap are subsequently *COMBINE*d into $\langle \xi, \eta \rangle$. Choose $\langle \phi, \psi \rangle = \langle \zeta_i, \zeta_{i+1} \rangle$ with the largest i such that $size(\langle \xi, \zeta_i \rangle) \le 2^{k-1}$. From $size(\langle \zeta_{i+1}, \eta \rangle) > 2^{k-1}$ it follows that $size(\langle \xi, \zeta_{i+1} \rangle) \le 2^{k-1}$, hence a larger i could have been chosen.

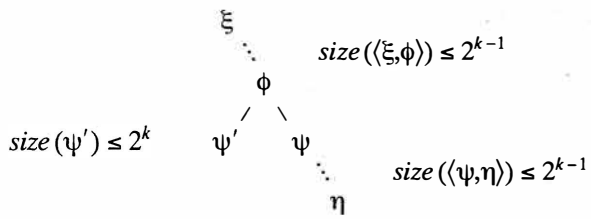


Figure 15. Claim A

From the induction hypothesis we find that $\langle \xi, \phi \rangle$, $\langle \psi, \eta \rangle$ have been proposed after step $k-1$; ψ' has been recognized after the *RECOGNIZE* in step k. Then $\langle \phi, \psi \rangle$ is *PROPOSE*d in step k and two *COMBINE* operations yield *proposed* $(\langle \xi, \eta \rangle)$.

(I)$_{k+1}$. Given (II)$_{k-1}$, (I)$_k$, we prove (I)$_{k+1}$. Let ξ be recognizable, $2^k < size(\xi) \le 2^{k+1}$.

- **Claim B**. There is an η with a pair of sons θ, ζ such that $size(\langle \xi, \eta \rangle) \le 2^k$, $size(\theta) \le 2^k$, $size(\zeta) \le 2^k$ and η, θ, ζ are recognizable.
  Proof. let ϕ_1 be the largest son of ξ, ϕ_2 the largest son of ϕ_1, etc. Let ϕ_j be the first one with size $\le 2^k$. Then $\eta = \phi_{j-1}$.

If $\eta = \xi$, (I)$_{k+1}$ follows trivially. Otherwise, Claim A holds and we find a situation as shown in Figure 16.

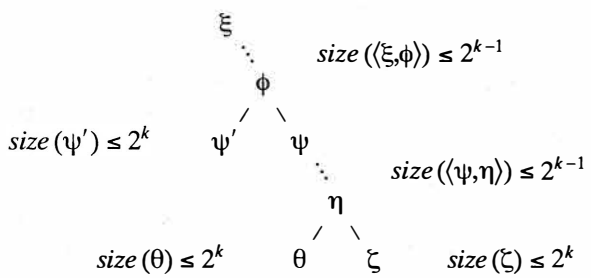


Figure 16. Claims B and A

From the induction hypothesis we find that $\langle \xi, \phi \rangle$, $\langle \psi, \eta \rangle$ have been proposed after step $k-1$; ψ', θ, ζ have been recognized after the *RECOGNIZE* in step k. Then $\langle \phi, \psi \rangle$, $\langle \eta, \zeta \rangle$ are *PROPOSE*d in step k and $\langle \xi, \psi \rangle$, $\langle \psi, \zeta \rangle$ are *COMBINE*d. The second *COMBINE* in step k proposed $\langle \xi, \zeta \rangle$. Hence ξ will be recognized in step $k+1$.

Conclusion. Thus we have established invariants (I) and (II). The "if" parts of the Theorem follow from (I), (II); the "only if" parts from the soundness of each of the operators *INITIALIZE*, *RECOGNIZE*, *PROPOSE* and *COMBINE*. □

A recognizing network

We define a connectionist network in a way that resembles the parsing network of Fanty (1985). The network consists of simple units computing AND and OR functions. The output of every unit is either 1 or 0. An AND unit is activated — i.e. its outputs have value 1 — iff all its inputs have value one; an OR unit is activated iff at least one of its inputs has value 1. In neural networks terminology: an AND unit with k inputs has a threshold value $k - 0.5$, an OR unit has a threshold value 0.5, irrespective of its number of inputs. In order to make a distinction between the two types of units we will write OR units between parentheses "( )" and AND units between brackets "[ ]".

For each triangle $\langle A, i, j \rangle \in \Xi$, the network contains a unit $(R\langle A, i, j \rangle)$ with an activation level corresponding to the value of $recognized(\langle A, i, j \rangle)$. Likewise, $proposed(\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle)$ is represented by a unit $(\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle)$. Furthermore, we need an output unit (*accept*) that is activated only if the sentence is accepted and input units $(\langle a, i \rangle)$ for $a \in \Sigma \cup \{\$\}$, $1 \le i \le n+1$. It is assumed that the input units are activated externally and that their activation level remains fixed. If a sentence has m words, then unit $(\langle m+1, \$ \rangle)$ should be activated to mark the end of the sentence.

- *INITIALIZE* is implemented by linking the units $(\langle a, i \rangle)$ to $(\langle A, i-1, i \rangle)$ for $A \to a \in P$.
- For the *PROPOSE* operation, for all $A \to BC \in P$ and $0 \le i < k < j \le n$, a link from $(R\langle B, i, k \rangle)$ to $(\langle \langle A, i, j \rangle, \langle C, k, j \rangle \rangle)$ and a link from $(R\langle C, k, j \rangle)$ to $(\langle \langle A, i, j \rangle, \langle B, i, k \rangle \rangle)$ are added to the network.
- For an implementation of *RECOGNIZE*, we need additional match units $[\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle]$ for each $\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle \in \Gamma$. This is because a unit $(R\langle A, i, j \rangle)$ can be recognized in more than one way. It should be recognized if both $(\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle)$ and $(R\langle B, k, l \rangle)$ are active for some $\langle B, k, l \rangle \in \Xi$. To this end, $(R\langle B, k, l \rangle)$ and $(\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle)$ are linked to $[\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle]$ that ANDs their values; $[\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle]$ is linked to $(R\langle A, i, j \rangle)$.
- For the *COMBINE* operation, we also need additional match units. For each $\langle \langle A, i, j \rangle, \langle B, k, l \rangle \rangle$ and $\langle \langle B, k, l \rangle, \langle C, m, n \rangle \rangle \in \Gamma$, an AND unit $[\langle \langle A, i, j \rangle, \langle B, k, l \rangle, \langle C, m, n \rangle \rangle]$ is added. It receives

input from $(\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle)$ and $(\langle\langle B,k,l\rangle,\langle C,m,n\rangle\rangle)$ and sends output to $(\langle\langle A,i,j\rangle,\langle C,m,n\rangle\rangle)$.

- The (*accept*) unit receives input from match units $[accept, i]$ that will be activated if a sentence of length i could be recognized. This is accomplished by linking $(\langle \$, i+1\rangle)$ and $(R\langle S, 0, i\rangle)$ to $[accept, i]$.

An example of a small fraction of the network is given in Figure 17. It represents the units that are used for the recognition of the propositional phrase $\langle PP, 5, 8\rangle$.

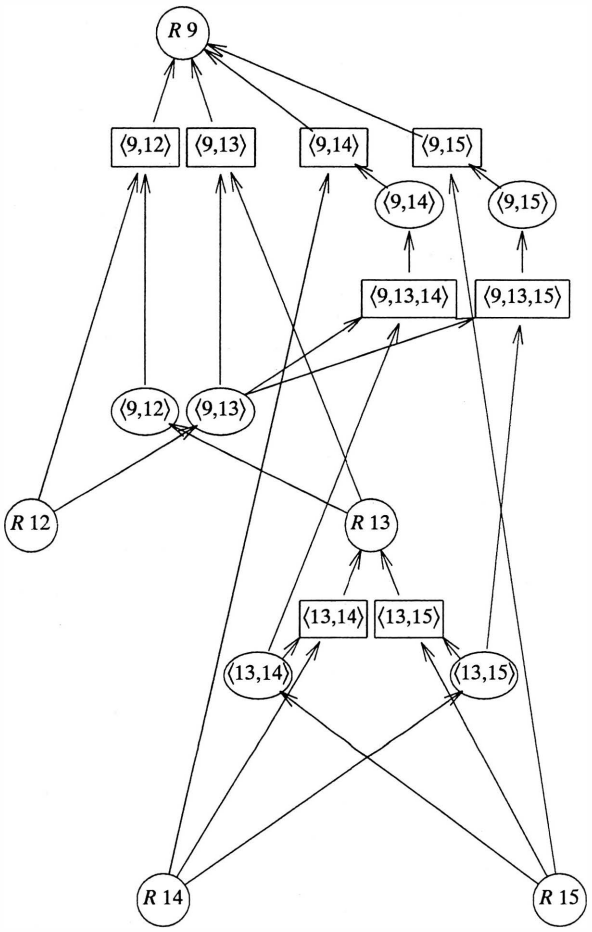


Figure 17. A fraction of the recognizing network

Construction of the shared forest

The main purpose of modifying Rytter's algorithm is the introduction of invariant (II). It will become clear now why we need it. Tacitly we have done all the necessary preparations for the extension to a parsing algorithm, all that is left is to reap the results.

Let $\langle A,i,j\rangle$ be parsable for a particular input string $a_1 \cdots a_m$ $(m \le n)$, and assume $\langle A,i,j\rangle \neq \langle S,0,m\rangle$. Then the following two conditions hold:

(i) $A \Rightarrow^+ a_{i+1} \cdots a_j$,

(ii) $S \Rightarrow^+ a_1 \cdots a_i A a_{j+1} \cdots a_m$.

In other words, $\langle A,i,j\rangle$ is parsable if $\langle A,i,j\rangle$ is recognizable and $\langle\langle S,0,m\rangle,\langle A,i,j\rangle\rangle$ is proposable. Consequently, *parsed*$(\langle A,i,j\rangle)$ must be made *true* if both *proposed*$(\langle\langle S,0,m\rangle,\langle A,i,j\rangle\rangle)$ and *recognized*$(\langle A,i,j\rangle)$ are *true*. This can be done in parallel in one step! We define an additional boolean table *parsed*$(\langle A,i,j\rangle)$ for $\langle A,i,j\rangle \in \Xi$ and an operation on Ξ, Γ and the table *parsed*, as follows:

PARSE :
for all $\langle A,i,j\rangle \in \Xi$
do *parsed*$(\langle A,i,j\rangle)$:= *false* **od** ;
if *recognized*$(\langle S,0,m\rangle)$
then *parsed*$(\langle S,0,m\rangle)$:= *true* ;
for all $\langle A,i,j\rangle \in \Xi$
do **if** *proposed*$(\langle\langle S,0,m\rangle,\langle A,i,j\rangle\rangle)$
and *recognized*$(\langle A,i,j\rangle)$
then *parsed*$(\langle A,i,j\rangle)$:= *true* **fi**
od
fi

The recognition algorithm is extended to a full-fledged parsing algorithm by inserting one *PARSE* operation after the **repeat** loop.

We can extend the network accordingly. The table *parsed* will be represented by a collection of AND units $[P\langle A,i,j\rangle]$. Additionally, we need a collection of match units $[Q_m\langle A,i,j\rangle]$ for $1 \le m \le n$ and $\langle A,i,j\rangle \in \Xi$ and a collection of OR match units $(Q\langle A,i,j\rangle)$ for $\langle A,i,j\rangle \in \Xi$.

- $[Q_m\langle A,i,j\rangle]$ will be activated if *parsed*$(\langle S,0,m\rangle)$ = *true* and *proposed*$(\langle\langle S,0,m\rangle,\langle A,i,j\rangle\rangle)$ = *true*. That is, for all possible values of m, $[P\langle S,0,m\rangle]$ and $(\langle\langle S,0,m\rangle,\langle A,i,j\rangle\rangle)$ are linked to $[Q_m\langle A,i,j\rangle]$.
- $(Q\langle A,i,j\rangle)$ will be activated if the above holds for *some m*. To this end, each $[Q_m\langle A,i,j\rangle]$ is linked to $(Q\langle A,i,j\rangle)$.
- $[P\langle A,i,j\rangle]$, obviously, receives input from $(R\langle A,i,j\rangle)$ and $(Q\langle A,i,j\rangle)$.
- In order to start the parsing phase, all $[accept, m]$ units are linked to $(Q\langle S,0,m\rangle)$. If a string of length m is accepted, then $(R\langle S,0,m\rangle)$ will be active, hence $[P\langle S,0,m\rangle]$ will be activated.

If $(Q\langle A,i,j\rangle)$ is activated (via $[Q_l\langle A,i,j\rangle]$) by any $[P\langle S,0,l\rangle]$ with $1 \le l \le m$, it is also activated by $[P\langle S,0,m\rangle]$, because $\langle\langle S,0,m\rangle,\langle S,0,l\rangle\rangle$ and $\langle\langle S,0,l\rangle,\langle A,i,j\rangle\rangle$ can be *COMBINE*d. Thus $[P\langle A,i,j\rangle]$ will be activated if and only if $\langle A,i,j\rangle$ is parsable.

Complexity of the network

The number of input units ($\langle a, i\rangle$) is $(|\Sigma|+1)\cdot(n+1) = O(|\Sigma|\cdot n)$. The *COMBINE* match units $[\langle\langle A,i,j\rangle,\langle B,k,l\rangle,\langle C,m,n\rangle\rangle]$ account for the highest order of all other types of units, $O(|N|^3\cdot n^6)$, yielding a total of $O(|\Sigma|\cdot n + |N|^3\cdot n^6)$ units. It is easy to verify that the number of connections is also $O(|\Sigma|\cdot n + |N|^3\cdot n^6)$.

These numbers conform to the best known complexity measures for logarithmic parsing algorithms: $O(\log n)$ time on a CRCW PRAM and $O(\log^2 n)$ time on a CREW PRAM. PRAM models use $O(n^6)$ processors. It is not obvious that an equivalent network exists with the same order of complexity. A general method to construct a network composed of AND and OR units for an arbitrary PRAM is given by Stockmeyer and Vishkin (1984). Applying this general method, however, would yield $O(n^{13})$ units, rather than our custom-tailored network of $O(n^6)$ units.

Meta-parsing

We defined units for *all* $\langle A,i,j\rangle\in\Xi$ and $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle\in\Gamma$. A large fraction of these units will never be needed. For any particular grammar we can establish a much smaller network, by an algorithm that closely resembles the parsing algorithm. Such analysis has been called *meta-parsing* (Nijholt, 1990).

A triangle $\langle A,i,j\rangle$ is called *meta-recognizable* if $\langle A,i,j\rangle$ is recognizable for some input string $a_1\cdots a_m\in\Sigma^m$, $(m\le n)$. Similarly, $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle$ is called *meta-proposable* if there is an input string such that $\langle A,i,j\rangle$ is proposable; $\langle A,i,j\rangle$ is called *meta-parsable* if there is an input string such that $\langle A,i,j\rangle$ is parsable. These meta-properties can be computed in advance, and incorporated in the structure of the network. The meta-recognizable and meta-parsable items for our example grammar and $n=8$ are shown in tabular form in Figures 18 and 19. The meta-parsing algorithm is identical to the parsing algorithm but for two small differences:

- *meta-recognized*($\langle A,i-1,i\rangle$) is made *true* if $A\rightarrow a\in P$ *for any* $a\in\Sigma$,
- *meta-parsed*($\langle S,0,i\rangle$) is made *true for every* $\langle S,0,i\rangle$ *that has been meta-recognized.*

It is easy to verify that after application of the meta-algorithm, $\langle A,i,j\rangle$ has been recognized if and only if $\langle A,i,j\rangle$ is meta-recognizable; similarly for meta-proposable and meta-parsable items.

For the construction of the shared forest, we only have to consider triangles that are meta-parsable. All triangles that are not meta-parsable can be discarded:

*d *n *v *p	NP	VP PP		S NP	VP PP		S NP
	*d *n *v *p	NP	VP PP		S NP	VP PP	
		*d *n *v *p	NP	VP PP		S NP	VP PP
			*d *n *v *p	NP	VP PP		S NP
				*d *n *v *p	NP	VP PP	
					*d *n *v *p	NP	VP PP
						*d *n *v *p	NP
							*d *n *v *p

Figure 18. $A\in t_{i,j}$ if $\langle A,i,j\rangle$ is meta-recognizable

*d	NP			S NP			S
	*n						
		*v *p		VP PP			VP
			*d	NP			NP
				*n			
					*v *p		VP PP
						*d	NP
							*n

Figure 19. $A\in t_{i,j}$ if $\langle A,i,j\rangle$ is meta-parsable

even if such a triangle is recognized, it can never contribute to a parse tree for any input. Thus we define the *minimal* set of triangles and triangles with gaps, as follows:

$$\Xi_{\min} = \{\langle A,i,j\rangle\in\Xi \mid \langle A,i,j\rangle \text{ is meta-parsable}\},$$

$$\Gamma_{\min} = \{\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle\in\Gamma \mid \langle A,i,j\rangle\in\Xi_{\min},\ \langle B,k,l\rangle\in\Xi_{\min},\ \langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle \text{ is meta-proposable}\}.$$

While constructing the network, we only have to introduce units for $\langle A,i,j\rangle\in\Xi_{\min}$, $\langle\langle A,i,j\rangle,\langle B,k,l\rangle\rangle\in\Gamma_{\min}$ and appropriate match units. The reduced network still yields the shared forest.

Robustness of the network

In contrast with Fanty's network, even the minimal network is rather robust. When a few units do not function, it is most likely that the proper input strings will be accepted. There is a multitude of different ways in which a triangle can be recognized; if the most direct path is broken, chances are that the triangle is recognized by an alternative path, using slightly more time. That is, unless one of the relatively few vital units breaks down, the recognition network shows graceful degradation. The parsing part of the network has no redundancy, however. If any unit fails, a triangle in the shared forest may be lost. But this is less dramatic than failure to recognize a valid sentence.

It is possible to supplement the recognition network with a robust parsing network if a top-down structure is used that is equivalent to the bottom-up structure, as in Fanty's network. Such a top-down network would yield a parse forest in logarithmic, rather than constant time. But that does not really matter as time complexity of the network is logarithmic anyway.

Bibliographic notes

The CYK algorithm can be found in any textbook on formal languages, e.g. (Harrison, 1978). A connectionist network for the CYK algorithm has been defined by Fanty (1985) and circulated on a wider scale in (Fanty, 1986).

Rytter's recognition algorithm is presented in (Rytter, 1985) and (Gibbons and Rytter, 1988). A similar algorithm is independently described by Brent and Goldschlager (1984). The operators *PROPOSE*, *COMBINE* and *RECOGNIZE* were called *ACTIVATE*, *SQUARE* and *PEBBLE* in the original algorithm. The word "activate" had to be changed so as to avoid confusion with activation of a unit. The new identifiers are chosen because we operate in a parsing context ("recognize") rather then a combinatorial context ("pebble"). Rytter's algorithm performs the following steps:

- step 0: *INITIALIZE*
- step k ($k > 0$): *ACTIVATE*; *SQUARE*; *SQUARE*; *PEBBLE*

which do *not* satisfy invariant (II)! Hence the algorithm does not allow a similar trivial extension for the computation of a shared forest. In (Gibbons and Rytter, 1988), the correctness of the Rytter's algorithm is derived from the correctness of a "pebble game" on binary trees, which has a rather complicated proof. The proof of the modified algorithm as presented above is a lot simpler, mainly due to the introduction of invariant (II).

EXTENSION TO ARBITRARY CONTEXT-FREE GRAMMARS

It is possible to define a similar parsing network for an arbitrary context-free grammar. Rytter's algorithm can be regarded as a speed-up of the CYK algorithm, using more resources. In the same way, bottom-up versions of Earley's algorithm (Graham, Harrison and Ruzzo, 1980), (Chiang and Fu, 1984) can be speeded up in a similar way. Triangles have the form $\langle A \rightarrow \alpha \bullet \beta, i, j \rangle$ for $A \rightarrow \alpha\beta \in P$ and $0 \leq i \leq j \leq n$.

$A \rightarrow \alpha \bullet \beta$ is recognizable iff $\alpha \Rightarrow^* a_{i+1} \cdots a_j$. Proposability can be defined accordingly. A triangle $\langle A \rightarrow \alpha \bullet \beta, i, j \rangle$ is parsable if there is a $\gamma \in V^*$ such that $S \Rightarrow^* a_1 \cdots a_i A \gamma$, $\alpha \Rightarrow^* a_{i+1} \cdots a_j$ and $\beta\gamma \Rightarrow^* a_{j+1} \cdots a_m$.

The network for arbitrary CFGs has $O(g^3 \cdot |P|^3 \cdot n^6)$ units and $O(g^3 \cdot |P|^3 \cdot n^6)$ connections, in which g is the average number of symbols in the right-hand side of a grammar rule. For a full treatment we refer to (Sikkel and Nijholt, 90).

A similar parsing algorithm for arbitrary CFGs on PRAM models is discussed in (de Vreught and Honig, 1991).

CONCLUSIONS

A modification of Rytter's logarithmic time recognition algorithm for CNF grammars has been introduced. This algorithm is conceptually easier than the original, and the correctness proof is a great deal simpler. Furthermore, the construction of a shared parse forest represented by a set of triangles can be added in constant time.

We have defined a connectionist network that parses a CNF grammar with the above algorithm in $O(\log n)$ time using $O(|\Sigma| \cdot n + |N|^3 \cdot n^6)$ units. This conforms to the best known complexity bounds on a CRCW PRAM, and is a factor $\log n$ faster than the best algorithm on a CREW PRAM known to date. A Similar network can be constructed for an arbitrary context-free grammar.

A network of minimum size for a particular grammar can be custom-tailored. The meta-parsing algorithm that establishes the configuration of a network for the specific grammar is almost identical to the parsing algorithm that is implemented by the network.

The network is robust in the sense that a few broken down units will most likely cause some degradation in performance but still all valid sentences will

be recognized.

A network structure with $O(n^6)$ units is too large for any serious practical implementation in natural language processing. The purpose of our investigations, however, has been to push the time complexity to its very limits to see how much parallelism is possible *in principle*. These results confirm that connectionist networks can be used as a suitable abstract machine model for parallel algorithms. It is also confirmed that traditional parsing algorithms for general context-free languages can be given connectionist implementations, allowing integration into more comprehensive connectionist networks for natural language analysis.

REFERENCES

Brent, Richard P. and Goldschlager, Leslie M. (1984). "A Parallel Algorithm for Context-Free Parsing," *Australian Computer Science Communications* **6**, 7, 7.1–7.10.

Chiang, Y.T. and Fu, K.S. (1984). "Parallel Parsing Algorithms and VLSI implementations for Syntactic Pattern Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-6**, 3, 302–314.

Cottrell, Garrison W. (1989). *"A Connectionist Approach to Word Sense Disambiguation"*, Research Notes in Artificial Intelligence, Morgan Kaufmann Publishers.

Fanty, Mark A. (1985). *"Context-free Parsing in Connectionist Networks"*, report TR 174, Computer Science Dept., University of Rochester, Rochester, NY.

Fanty, Mark A. (1986). M.A. Fanty: "Context-free Parsing in Connectionist networks," in: J.S. Denker (Ed.), *"Neural Networks for Computing"*, Snowbird, UT, API conference proceedings 151, American Institute of Physics, 140–145.

Gibbons, Alan and Rytter, Wojciech (1988). *"Efficient Parallel Algorithms"*, Cambridge University Press.

Graham, Susan L, Harrison, Michael A. and Ruzzo, Walter L. (1980). "An Improved Context-Free Recognizer," *ACM Transactions on Programming Languages and Systems* **2** 415–462.

Harrison, Michael (1978). *"Introduction to Formal Language Theory"*, Addison-Wesley, Reading, Mass.

Howells, Tim (1988). "VITAL: a Connectionist Parser," *Proc. 10th Conf. of the Cognitive Science Society* 18–25.

Nakagawa, Hiroshi and Mori, Tatsunori (1988). "A parser based on a connectionist model," *Proc. 12th Int. Conf. on Computational Linguistics* (COLING'88), Budapest 454–458.

Nijholt, Anton (1990). "Meta-Parsing in Neural Networks," *Proc. 10th European Meeting on Cybernetics and Systems Research*, R. Trappl (Ed.), Vienna 969–976.

Rytter, Wojciech (1985). "On the recognition of context free languages," in: *"Proc. 5th Symp. on Fundamentals of Computation Theory"*, Lecture Notes in Computer Science 208, Springer-Verlag 315–22.

Selman, Bart and Hirst, Graeme (1987). "Parsing as an energy minimization problem," in: *Genetic algorithms and Simulated Annealing*, Research Notes in AI, Morgan Kaufmann Publishers, Los Altos 141–154.

Sikkel, Klaas (1990). *"Connectionist Parsing of Context-Free Languages"*, Memoranda Informatica 90-30, Computer Science Department, University of Twente, Enschede, The Netherlands.

Stockmeyer, Larry and Vishkin, Uzi (1984). Simulation of Parallel Random Access Machines by Circuits, *SIAM J. Comput.*, **13**, 2, 409–422.

de Vreught, Hans and Honig, Job (1991) "Slow and Fast Parallel Recognition," *Proc. 2nd Int. Workshop on Parsing Technologies*, (IWPT'91), Cancun, Mexico.

Waltz, David L. and Pollack, Jordan B. (1988). "Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation," in: D.L. Waltz, J.A. Feldman (Eds.), *Connectionist models and their implications*, Readings from Cognitive Science, Ablex Publishing Co., Norwood, NJ, 181–204.

SLOW AND FAST PARALLEL RECOGNITION

Hans de Vreught, Job Honig*
Delft University of Technology
Faculty of Technical Mathematics and Informatics
Section Theoretical Computer Science
Julianalaan 132, 2628 BL Delft, The Netherlands
e-mail: hdev@dutiae.tudelft.nl, joho@dutiae.tudelft.nl

ABSTRACT

In the first part of this paper a slow parallel recognizer is described for general CFG's. The recognizer runs in $\Theta(n^3/p(n))$ time with $p(n) = O(n^2)$ processors. It generalizes the items of the Earley algorithm to double dotted items, which are more suited to parallel parsing. In the second part a fast parallel recognizer is given for general CFG's. The recognizer runs in $O(\log\ n)$ time using $\Theta(n^6)$ processors. It is a generalisation of the Gibbons and Rytter algorithm for grammars in CNF.

1 INTRODUCTION

The subject of context-free parsing is well studied, e.g. see (Aho and Ullman, 1972; 1973; Harrison, 1978). Nowadays, research on the subject has shifted to parallel context-free parsing (op den Akker, Alblas, Nijholt, and Oude Luttighuis, 1989). Two areas of interest can be distinguished: slow and fast parallel parsing. We call a parallel algorithm fast when it does its job in polylogarithmic time. This is in contrast to the sequential case, in which algorithms are called fast when they run in polynomial time. Obtaining a fast parallel algorithm is often quite simple: when the fast sequential algorithm is highly parallelizable, using an exponential number of processors is sufficient. This is not very realistic, however.

A parallel algorithm is called feasible only when it uses a polynomial number of processors. Note that when a feasible slow parallel algorithm runs in polynomial time, it can be simulated by a fast sequential algorithm. Therefore in practice we often see that slow parallel is fast enough; fast parallel algorithms often achieve their speed because of their huge number of processors and large amounts of storage.

Several authors have studied algorithms for slow parallel recognition. Most of these algorithms are variants of the Cocke-Younger-Kasami (CYK) algorithm and the Earley algorithm. In the first part of this paper another slow parallel recognizer is given (de Vreught and Honig, 1989; 1990b). Its new feature is that it uses double dotted items, which are more natural for parallel parsing; these items make it easy to do error determination, a feature that is shared with most parallel bottom up algorithms. Although there are some similarities between the three algorithms, they should not be regarded as variants of each other since they all fill their respective matrices with different 'items' and for entirely different reasons.

When compared to a parallel version of the Earley algorithm, which would have to be bottom up, our algorithm generates far less items on the principal diagonal of the recognition matrix. A detailed comparison of the items required by the given algorithm and the Earley algorithm will be necessary to show the strengths or weaknesses of both approaches to parallel parsing. The sizes of the item sets in relation to particular classes of grammars is still under research.

The subject of fast parallel parsing is relatively new. Amongst the first to give a fast parallel recognizer were Gibbons and Rytter (1988). Their recognizer requires a grammar in CNF; it can be regarded as the fast par-

*Using initials: J.P.M. de Vreught and H.J. Honig.

allel version of the slow parallel CYK algorithm. The speeded up version is obtained by also examining the consequences of incomplete items. When an incomplete item gets completed, we can also complete the consequences immediately. The reason for the algorithm being fast is based on the fact that for every skewed tree (with n internal nodes) of height $O(n)$ describing the composition of a certain item, there exists a reasonably well balanced one of height $O(\log\ n)$ that uses both complete and incomplete items.

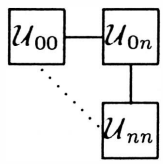
In the second part of the paper a fast parallel recognizer for general CFG's is given (de Vreught and Honig, 1990a). In spite of the fact that any CFG can be transformed into CNF in $O(1)$ time, using CNF is undesirable in practice (especially in natural language processing). The fast parallel recognizer does not need to transform the grammar. The fast parallel recognizer can be regarded as the fast parallel version of the slow parallel recognizer described in the first part. The fast parallel recognizer is based on the Gibbons and Rytter algorithm for grammars in CNF (Gibbons and Rytter, 1988). The paper is concluded with some final remarks.

2 THE SLOW PARALLEL RECOGNIZER

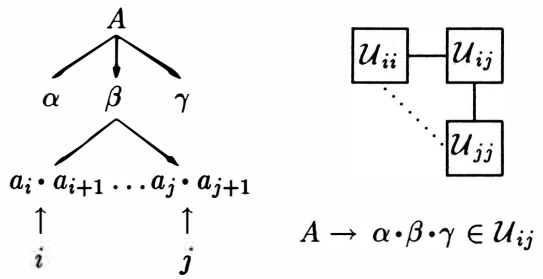
We start by sketching the ideas behind the slow parallel recognizer. Then we will give an inductive relation which plays a central role in our algorithms. Finally we will present the slow parallel recognizer.

2.1 INFORMAL DESCRIPTION

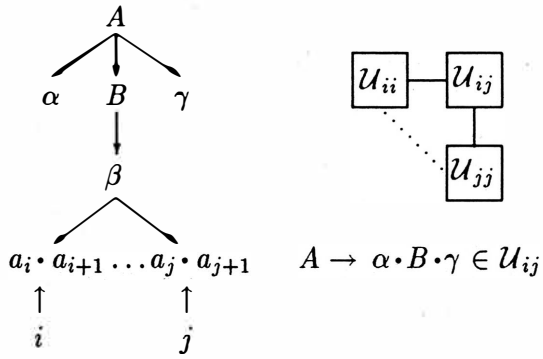
Let $a_1 \ldots a_n$ be the string to be recognized. We are going to build an upper triangular matrix $\mathcal{U}$ as shown below.



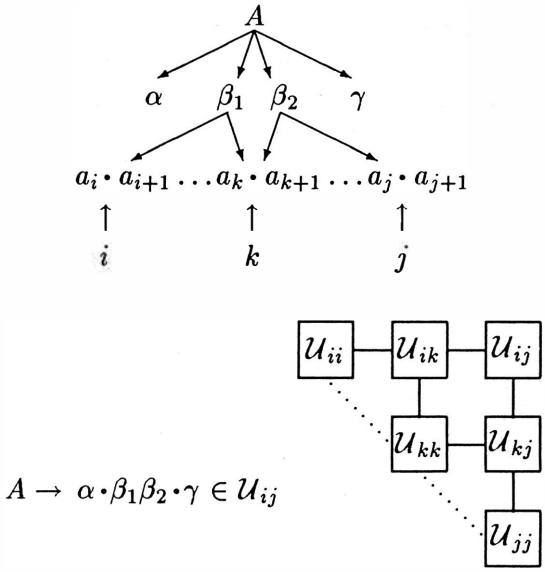
In each cell $\mathcal{U}_{ij}$ we enter items of the form $A \rightarrow \alpha \cdot \beta \cdot \gamma$ such that $A \rightarrow \alpha\beta\gamma$ is a production and $\beta \Rightarrow^* a_{i+1} \ldots a_j$. We will also insist that if $\beta = \lambda$ then $\alpha\gamma = \lambda$:



Suppose $B \rightarrow \cdot\beta\cdot \in \mathcal{U}_{ij}$ and let $A \rightarrow \alpha B \gamma$ be a production. In that case we can assert that $A \rightarrow \alpha \cdot B \cdot \gamma \in \mathcal{U}_{ij}$:

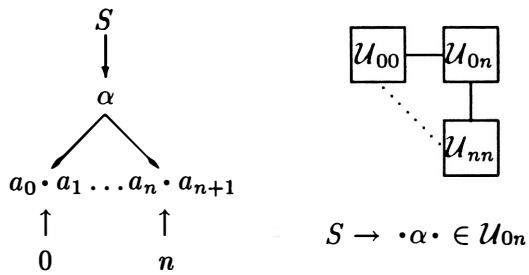


This assertion follows from the application of the inclusion operation to $B \rightarrow \cdot\beta\cdot \in \mathcal{U}_{ij}$. Another operation is concatenation. If $A \rightarrow \alpha \cdot \beta_1 \cdot \beta_2 \gamma \in \mathcal{U}_{ik}$ and $A \rightarrow \alpha\beta_1 \cdot \beta_2 \cdot \gamma \in \mathcal{U}_{kj}$ then we can assert that $A \rightarrow \alpha \cdot \beta_1\beta_2 \cdot \gamma \in \mathcal{U}_{ij}$, applying the concatenation operation:



When the entries of matrix $\mathcal{U}$ are closed with respect to the operations, we look for an item

$S \rightarrow \cdot\alpha\cdot \in \mathcal{U}_{0n}$ where S is the start symbol of the grammar:



In the following, we will give a relation **U** defining the item sets constructed during the recognition process. We do this by identifying the matrix $\mathcal{U}$ with the relation **U** such that $A \rightarrow \alpha\cdot\beta\cdot\gamma \in \mathcal{U}_{ij}$ iff $(i,j,A \rightarrow \alpha\cdot\beta\cdot\gamma) \in \mathbf{U}$.

2.2 THE RELATION

Let $G = (V, \Sigma, P, S)$ be the CFG in question and let $x = a_1 \ldots a_n$ the string to be recognized. Furthermore, let $\cdot \notin V$ and let λ be the empty string. Finally, let $\mathcal{J} = \{0,\ldots,n\}^2 \times \{A \rightarrow \alpha\cdot\beta\cdot\gamma \mid A \rightarrow \alpha\beta\gamma \in P\}$.

Definition 2.2.1 $\mathbf{U} = \{(i,j,A \rightarrow \alpha\cdot\beta\cdot\gamma) \in \mathcal{J} \mid A \Rightarrow \alpha\beta\gamma$ and $\beta \Rightarrow^* a_{i+1}\ldots a_j$ and if $\beta = \lambda$ then $\alpha\gamma = \lambda\}$

In (de Vreught and Honig, 1989) some variants of **U** are examined; for instance, one of them takes context into account. The disadvantage of definition 2.2.1 is that it is not immediately clear how to determine whether or not an item is in the relation. For this purpose we need an inductive definition.

Definition 2.2.2 The relation **U'** over $\mathcal{J}$ is defined as follows:

- If $A \rightarrow \lambda \in P$ then $(j,j,A \rightarrow \cdot\cdot) \in \mathbf{U}'$ for any $j \in \{0,\ldots,n\}$.
  This item is a base item.
- If $A \rightarrow \alpha a_j \gamma \in P$ then $(j-1,j,A \rightarrow \alpha\cdot a_j\cdot\gamma) \in \mathbf{U}'$ for any $j \in \{1,\ldots,n\}$.
  This item is a base item.
- If $A \rightarrow \alpha B\gamma \in P$ and $(i,j,B \rightarrow \cdot\beta\cdot) \in \mathbf{U}'$ then $(i,j,A \rightarrow \alpha\cdot B\cdot\gamma) \in \mathbf{U}'$.
  This operation is called inclusion.
- If $(i,k,A \rightarrow \alpha\cdot\beta_1\cdot\beta_2\gamma) \in \mathbf{U}'$ and $(k,j,A \rightarrow \alpha\beta_1\cdot\beta_2\cdot\gamma) \in \mathbf{U}'$ then $(i,j,A \rightarrow \alpha\cdot\beta_1\beta_2\cdot\gamma) \in \mathbf{U}'$.
  This operation is called concatenation.
- Nothing is in **U'** except those elements which must be in **U'** by applying the preceding rules finitely often.

It can be proved that $\mathbf{U} = \mathbf{U}'$.

2.3 THE RECOGNIZER

We will now present the recognition algorithm (de Vreught and Honig, 1989). In the algorithm **mode** is either **sequence** or **parallel**.

```
Recognizer(n):
    for i := 0 to n do
        for j := 0 to n − i in parallel do
            case i
                = 0:
                    Empty(j + i)
                = 1:
                    Symbol(j + i)
                > 1:
                    U_{j,j+i} := ∅
                    for k := 1 to i − 1
                    in mode do
                            Concatenator
                                (j, j + k, j + i)
            while U_{j,j+i} still changes do
                Concatenator(j, j, j + i)
                Concatenator(j, j + i, j + i)
                Includer(j, j + i)
    return Test(n)

Empty(j):
    U_{jj} := {A → ·· | A → λ ∈ P}

Symbol(j):
    U_{j−1,j} := {A → α·a_j·γ | A → αa_jγ ∈ P}

Includer(i, j):
    for all B → ·β· ∈ U_{ij} do
        U_{ij} := U_{ij} ∪
            {A → α·B·γ | A → αBγ ∈ P}

Concatenator(i, k, j):
    for all A → α·β_1·β_2γ ∈ U_{ik}
    with | β_1 | = 1 do
        for all A → αβ_1·β_2·γ ∈ U_{kj} do
            U_{ij} := U_{ij} ∪ {A → α·β_1β_2·γ}
```

```
Test(n):
    accept := False
    for all S → α ∈ P do
        if S → •α• ∈ U_0n then
            accept := True
    return accept
```

Although the algorithm fills the matrix diagonal by diagonal, there are many other filling orders for the matrix (de Vreught and Honig, 1989). Note that all cells on a diagonal can be filled independently of each other. When **mode = sequence**, it can be shown that a CREW-PRAM (Concurrent Read Exclusive Write - Parallel RAM) (Quinn, 1987) with $p(n) = O(n)$ processors can fill the matrix in $T(n) = \Theta(n^3/p(n))$ time.

The concatenations done in the loop over k in Recognizer can also be done independently of each other. However, in that case the architecture must allow parallel writing in cell $U_{j,j+i}$. Thus when **mode = parallel**, it can be shown that a CRCW-PRAM (Concurrent Read Concurrent Write - Parallel RAM) (Quinn, 1987) with $p(n) = O(n^2)$ processors can fill the matrix in $T(n) = \Theta(n^3/p(n))$ time. In both cases the space complexity is dominated by the matrix: $S(n) = O(n^2)$ space.

Example 2.3.1 Consider the string *aabcc* and the CFG $G = (V, \Sigma, P, A)$:

- $V = \{A, B\} \cup \Sigma$
- $\Sigma = \{a, b, c\}$
- P contains the following productions:
  - $A \rightarrow aB$
  - $B \rightarrow Acc$
  - $B \rightarrow b$

Notice that G is λ-free (this simplifies the example). From the given grammar and string, the following matrix (see figure 1) can be obtained.

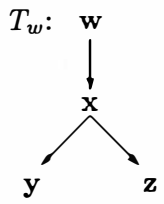
3 THE FAST PARALLEL RECOGNIZER

In this section we will sketch the ideas behind the fast algorithm. The proof that the recognizer is fast uses a pebble game, described in (Gibbons and Rytter, 1988), and critically depends on the fact that the 'minimal composition trees' are linear in size (with respect to the length of the string to be recognized). Instead of determining **U** directly we will compute its extension $\tilde{\mathbf{U}}$, on which the fast parallel recognizer is based. Finally we will describe the recognizer for a general CFG. The algorithms is based on the fast parallel Gibbons and Rytter recognizer for CFG's in CNF (Gibbons and Rytter, 1988).

3.1 COMPOSITION TREES

Definition 2.2.2 offers a way of justifying the presence of an item x in **U**. A justification is a sequence of rules corresponding to a proof showing why $x \in \mathbf{U}$. Sometimes an item x can be justified in more than one way. We will consider justifications one at a time. A complete justification of an item x in **U** will be called a composition for x; such a composition can be represented by a composition tree T_x. The nodes in T_x are labelled with the items mentioned in the antecedents of the rules of definition 2.2.2 that are applied; the root is labelled x.

Example 3.1.1 Suppose w is the result of an inclusion of x, x is the result of a concatenation of y and z, and y and z are base items. The composition tree T_w for w is as given below.



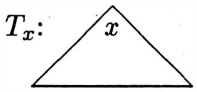
3.2 INFORMAL DESCRIPTION

We will speed up the slow parallel algorithm that computes relation **U** to a fast parallel algorithm computing **U** by using a relation denoted by $\tilde{\mathbf{U}}$ (given in section 3.4). The presence of each item x in **U** can be justified by means of a composition tree T_x. In T_x all

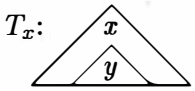
	$A \rightarrow \bullet a \bullet B$				$A \rightarrow \bullet aB \bullet$ $B \rightarrow \bullet A \bullet cc$
		$A \rightarrow \bullet a \bullet B$	$A \rightarrow \bullet aB \bullet$ $B \rightarrow \bullet A \bullet cc$	$B \rightarrow \bullet Ac \bullet c$	$B \rightarrow \bullet Acc \bullet$ $A \rightarrow a \bullet B \bullet$
			$B \rightarrow \bullet b \bullet$ $A \rightarrow a \bullet B \bullet$		
				$B \rightarrow A \bullet c \bullet c$ $B \rightarrow Ac \bullet c \bullet$	$B \rightarrow A \bullet cc \bullet$
					$B \rightarrow A \bullet c \bullet c$ $B \rightarrow Ac \bullet c \bullet$

Figure 1. Matrix $\mathcal{U}$ for *aabcc*

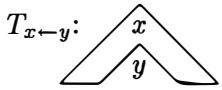
nodes are labelled with items in **U**. The root is labelled x. The other nodes are labelled by the items mentioned in the antecedents of the rules of the inductive definition of **U**. As an immediate consequence we have that each subtree of T_x is a composition tree too. We will represent T_x (or to be more exact: the existence of T_x) as in the figure below:



Suppose T_y exists. Thus we assume $y \in$ **U**. Let us see what the consequences of this assumption are. Suppose we can derive T_x for item x from T_y:

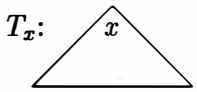


Assume we don't now wether or not y actually is in **U**. Instead of saying that we have determined T_x, we say that we have determined T_x except for the part T_y: we have the partial composition tree $T_{x \leftarrow y}$ (or better: its existence) represented as given below:

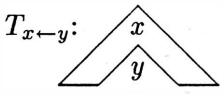


Note that $T_{x \leftarrow y}$ might exist whilst T_y does not (because $y \notin$ **U**). By using these partial composition trees, we draw conclusions from facts yet to be established. This makes the algorithm for the recognizer fast; the proof of this is based on Rytter's pebble game (Gibbons and Rytter, 1988).

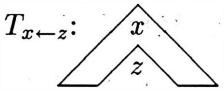
For each base item x (in **U**), we can assert the existence of a composition tree T_x:



Suppose x can be obtained from y by means of an inclusion operation. In that case we can assert the partial composition tree $T_{x \leftarrow y}$:

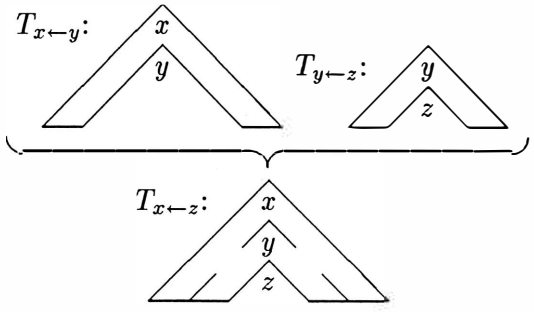


Now suppose that x can be obtained from y and z by means of a concatenation operation and assume that T_y exists (the case that T_z exists, is handled analogously). In that case we can assert the partial composition tree $T_{x \leftarrow z}$:

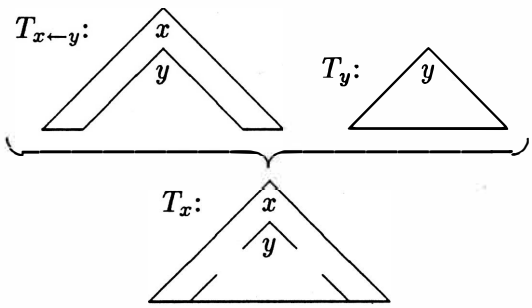


The rules for the inclusion and concatenation operations are called activation rules (the names of all rules are borrowed from the pebble game).

The square rule (a misnomer) merges two partial composition trees $T_{x \leftarrow y}$ and $T_{y \leftarrow z}$ to obtain the partial composition tree $T_{x \leftarrow z}$:



The final rule is the pebble rule, which merges a partial composition tree $T_{x \leftarrow y}$ and a composition tree T_y to obtain the composition tree T_x:



When we would define a composition tree for $\tilde{\mathbf{U}}$ in the same way as we did for $\mathbf{U}$, we would find that for an arbitrary $\mathbf{U}$-composition tree T_x there exists a reasonably well balanced $\tilde{\mathbf{U}}$-composition tree $\tilde{T}_x$, which also asserts that the item x is in $\mathbf{U}$. It can be shown that if the activation rule, the square rule, and the pebble rule are iterated $O(\log\ n)$ times, we have found the existence of at least one composition tree T_x for every x in $\mathbf{U}$ (and for only those). Therefore we can say that we can compute $\mathbf{U}$ in $O(\log\ n)$ time.

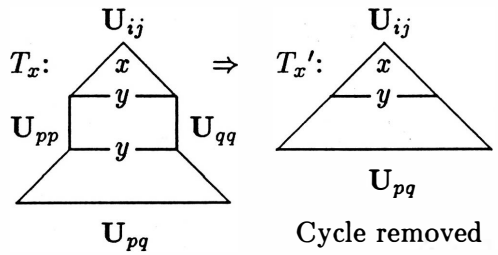
3.3 THE MINIMAL COMPOSITION SIZE

As a notational shortcut we will speak of an item x in $\mathbf{U}_{ij}$, by which we mean that $x \in \mathbf{U}$ and that x is of the form $(i, j, A \rightarrow \alpha \cdot \beta \cdot \gamma)$. The composition size will be defined as the number of operations in the composition tree. We call a composition tree minimal iff its composition size is minimal. In this section we will argue why the minimal composition size for item x in $\mathbf{U}_{ij}$ is linear in $j - i + 1$. There are two cases to consider:

- A composition tree which has an item appearing twice as a label on a path (such a tree is called a 'cyclic'[1] composition tree) is not minimal.
- An 'acyclic' composition tree has a linear composition size.

[1] A misnomer on our part.

Assume that for item x in $\mathbf{U}_{ij}$ we have found a cyclic composition tree T_x. So on a certain path in T_x we must have a certain item y in $\mathbf{U}_{pq}$ that appears twice as a label (the nontrivial path between those nodes is called a 'cycle'):



It is clear that when the part between the upper y and the lower y is removed from T_x, the number of operations in T_x' is less than the number in T_x. So after removing a cycle, we allways get a smaller composition tree. Thus the minimal composition tree is a member of the set of the acyclic composition trees.

We will now argue that any acyclic composition tree has a composition size bounded by a function linear in the length of the string to be recognized. Since we don't need a tight upper bound, we will not use an actual composition. Instead, we will assume that in every step on our way the worst case occurs. This may lead to a 'case' that is worse than the actual worst case.

We will assume that every internal node is the result of a concatenation. Suppose x is the result of an inclusion of y: in that case T_x contains one more operation than T_y. But when x is the result of a concatenation of y and z, then T_x contains one more operation than T_y and T_z together. Thus a concatenation can only lead to more (and never to fewer) operations than an inclusion. We will assume that the compositions are acyclic.

We define $M = |\{A \rightarrow \alpha \cdot \beta \cdot \gamma \mid A \rightarrow \alpha\beta\gamma \in P\}|$; M is an upper bound for the number of items in any $\mathbf{U}_{ij}$. Let us focus on an item x in $\mathbf{U}_{jj}$, see figure 2(a). We know that item x has an acyclic composition, so T_x is bounded in height by $O(M)$. Since a completely balanced tree has the maximum number of operations, we have an exponential number of

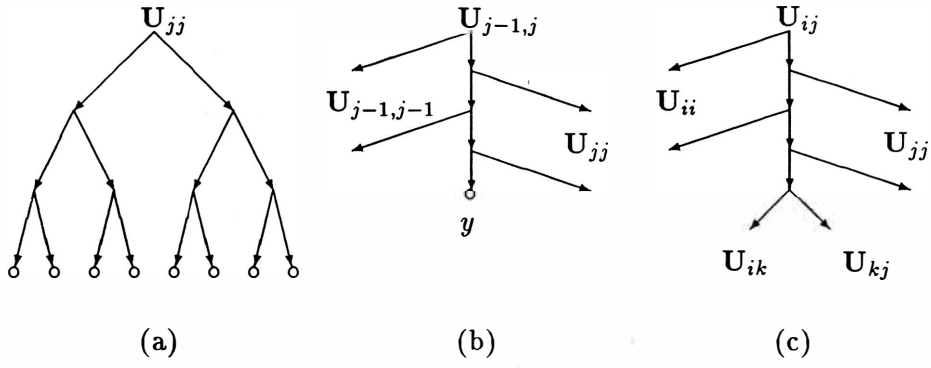


Figure 2. A simplified partial subtree of an acyclic T_x

operations in M. However, this number is independent of n. Thus there exist only $O(1)$ many operations in such a composition.

The next case is an item x in $\mathbf{U}_{j-1,j}$, see figure 2(b). We know that there must exist a path from x to a base item y in $\mathbf{U}_{j-1,j}$. All nodes on that path are in $\mathbf{U}_{j-1,j}$ and the path is bounded in length by $O(M)$. Any internal node on that path has one son in $\mathbf{U}_{j-1,j}$ and one son in either $\mathbf{U}_{j-1,j-1}$ or $\mathbf{U}_{jj}$ (if the node corresponds to an inclusion, this last son does not exist). Here too, it can be shown that only $O(1)$ operations are possible for item x.

The last case will be item x in $\mathbf{U}_{ij}$ with $i+1<j$, see figure 2(c). This is essentially like the previous case, but y is not a base item anymore. In this case y is the result of a concatenation of an item in $\mathbf{U}_{ik}$ and an item in $\mathbf{U}_{kj}$ with $i<k<j$. So instead we get $O(1)$ operations plus the number of operations needed for the item in $\mathbf{U}_{ik}$ and the item in $\mathbf{U}_{kj}$. These considerations lead to a difference equation, the solution of which shows that the number of operations for x is $O(n)$, see (de Vreught and Honig, 1990a).

3.4 THE EXTENDED RELATION

Definition 3.4.1 The relation $\tilde{\mathbf{U}}$ over $\mathcal{J} \cup \mathcal{J}^2$ is defined as follows:

- If $A \to \lambda \in P$ then $(j, j, A \to \cdot\cdot\ ) \in \tilde{\mathbf{U}}$ for any $j \in \{0, \ldots, n\}$.
  This rule is used for the initialization.
- If $A \to \alpha a_j \gamma \in P$ then $(j-1, j, A \to \alpha \cdot a_j \cdot \gamma\ ) \in \tilde{\mathbf{U}}$ for any $j \in \{1, \ldots, n\}$.
  This rule is used for the initialization.
- If $A \to \alpha B \gamma \in P$ and $B \to \beta \in P$ then $(i, j, A \to \alpha \cdot B \cdot \gamma\ ) \leftarrow (i, j, B \to \cdot \beta \cdot\ ) \in \tilde{\mathbf{U}}$ with $0 \le i \le j \le n$.
  This rule is called the activation rule for the inclusion operation.
- If $(i, k, A \to \alpha \cdot \beta_1 \cdot \beta_2 \gamma\ ) \in \tilde{\mathbf{U}}$ then $(i, j, A \to \alpha \cdot \beta_1 \beta_2 \cdot \gamma\ ) \leftarrow (k, j, A \to \alpha \beta_1 \cdot \beta_2 \cdot \gamma\ ) \in \tilde{\mathbf{U}}$ with $k \le j \le n$.
  This rule is called an activation rule for the concatenation operation.
- If $(k, j, A \to \alpha \beta_1 \cdot \beta_2 \cdot \gamma\ ) \in \tilde{\mathbf{U}}$ then $(i, j, A \to \alpha \cdot \beta_1 \beta_2 \cdot \gamma\ ) \leftarrow (i, k, A \to \alpha \cdot \beta_1 \cdot \beta_2 \gamma\ ) \in \tilde{\mathbf{U}}$ with $0 \le i \le k$.
  This rule is called an activation rule for the concatenation operation.
- If $x \leftarrow y \in \tilde{\mathbf{U}}$ and $y \leftarrow z \in \tilde{\mathbf{U}}$ then $x \leftarrow z \in \tilde{\mathbf{U}}$.
  This rule is called the square rule.
- If $x \leftarrow y \in \tilde{\mathbf{U}}$ and $y \in \tilde{\mathbf{U}}$ then $x \in \tilde{\mathbf{U}}$.
  This rule is called the pebble rule.
- Nothing is in $\tilde{\mathbf{U}}$ except those elements which must be in $\tilde{\mathbf{U}}$ by applying the preceding rules finitely often.

It can be shown that $\mathbf{U} = \tilde{\mathbf{U}} \cap \mathcal{J}$.

3.5 THE RECOGNIZER

We will present the fast parallel recognizer (de Vreught and Honig, 1990a).

Recognizer(n):
    $\tilde{\mathbf{U}} := \emptyset$
    for all i_1, i_2 such that $0 \leq i_1 \leq i_2 \leq n$
    in parallel do
        Initialization(i_1)
        ActivateInclusion(i_1, i_2)
    while $\tilde{\mathbf{U}}$ still changes do
        for all $i_1, \ldots, i_6$ such that
        $0 \leq i_1 \leq \ldots \leq i_6 \leq n$ **in parallel** do
            ActivateConcatenation($i_1, \ldots, i_3$)
            Square($i_1, \ldots, i_6$)
            Square($i_1, \ldots, i_6$)
            Pebble($i_1, \ldots, i_4$)
    return Test(n)

Initialization(j):
    $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
        $\{(j, j, A \rightarrow \bullet\bullet\ ) \in \mathcal{J} \mid A \rightarrow \lambda \in P\}$
    $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
        $\{(j-1, j, A \rightarrow \alpha \bullet a_j \bullet \gamma\ ) \in \mathcal{J} |$
        $A \rightarrow \alpha a_j \gamma \in P\}$

ActivateInclusion(i, j):
    $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
        $\{(i, j, A \rightarrow \alpha \bullet B \bullet \gamma\ ) \leftarrow$
        $(i, j, B \rightarrow \bullet \beta \bullet\ ) \in \mathcal{J}^2 |$
        $A \rightarrow \alpha B \gamma \in P$ and $B \rightarrow \beta \in P\}$

ActivateConcatenation(i, k, j):
    for all $A \rightarrow \alpha\beta_1\beta_2\gamma \in P$ do
        if $(i, k, A \rightarrow \alpha \bullet \beta_1 \bullet \beta_2 \gamma\ ) \in \tilde{\mathbf{U}}$ then
            $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
                $\{(i, j, A \rightarrow \alpha \bullet \beta_1 \beta_2 \bullet \gamma\ ) \leftarrow$
                $(k, j, A \rightarrow \alpha\beta_1 \bullet \beta_2 \bullet \gamma\ )\}$
            if $(k, j, A \rightarrow \alpha\beta_1 \bullet \beta_2 \bullet \gamma\ ) \in \tilde{\mathbf{U}}$ then
                $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
                    $\{(i, j, A \rightarrow \alpha \bullet \beta_1 \beta_2 \bullet \gamma\ ) \leftarrow$
                    $(i, k, A \rightarrow \alpha \bullet \beta_1 \bullet \beta_2 \gamma\ )\}$

Square($i_1, i_2, i_3, j_3, j_2, j_1$):
    for all $A_1 \rightarrow \alpha_1\beta_1\gamma_1$, $A_2 \rightarrow \alpha_2\beta_2\gamma_2$,
    $A_3 \rightarrow \alpha_3\beta_3\gamma_3 \in P$ do
        if $(i_1, j_1, A_1 \rightarrow \alpha_1 \bullet \beta_1 \bullet \gamma_1\ ) \leftarrow$
        $(i_2, j_2, A_2 \rightarrow \alpha_2 \bullet \beta_2 \bullet \gamma_2\ ) \in \tilde{\mathbf{U}}$
        and $(i_2, j_2, A_2 \rightarrow \alpha_2 \bullet \beta_2 \bullet \gamma_2\ ) \leftarrow$
        $(i_3, j_3, A_3 \rightarrow \alpha_3 \bullet \beta_3 \bullet \gamma_3\ ) \in \tilde{\mathbf{U}}$ then
            $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
                $\{(i_1, j_1, A_1 \rightarrow \alpha_1 \bullet \beta_1 \bullet \gamma_1\ ) \leftarrow$
                $(i_3, j_3, A_3 \rightarrow \alpha_3 \bullet \beta_3 \bullet \gamma_3\ )\}$

Pebble(i_1, i_2, j_2, j_1):
    for all $A_1 \rightarrow \alpha_1\beta_1\gamma_1$, $A_2 \rightarrow \alpha_2\beta_2\gamma_2 \in P$ do
        if $(i_1, j_1, A_1 \rightarrow \alpha_1 \bullet \beta_1 \bullet \gamma_1\ ) \leftarrow$
        $(i_2, j_2, A_2 \rightarrow \alpha_2 \bullet \beta_2 \bullet \gamma_2\ ) \in \tilde{\mathbf{U}}$
        and $(i_2, j_2, A_2 \rightarrow \alpha_2 \bullet \beta_2 \bullet \gamma_2\ ) \in \tilde{\mathbf{U}}$ then
            $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
                $\{(i_1, j_1, A_1 \rightarrow \alpha_1 \bullet \beta_1 \bullet \gamma_1\ )\}$

Test(n):
    accept := False
    for all $S \rightarrow \alpha \in P$ do
        if $(0, n, S \rightarrow \bullet\alpha\bullet\ ) \in \tilde{\mathbf{U}}$ then
            accept := True
    return accept

With the pebble game described in (Gibbons and Rytter, 1988), and the fact that the minimal composition size of an item is linear, we can show that any item can be constructed in $O(\log\ n)$ time. Thus $\mathbf{U} = \tilde{\mathbf{U}} \cap \mathcal{J}$ can be computed in $O(\log\ n)$ time. It can be shown that closure of $\tilde{\mathbf{U}}$ requires an extra $O(\log\ n)$ time following the computation of the completion of $\mathbf{U} \subseteq \tilde{\mathbf{U}}$ (detection of the closure of $\tilde{\mathbf{U}}$ is easy, whilst detection of the completion of $\mathbf{U} \subseteq \tilde{\mathbf{U}}$ is not).

It can be shown (de Vreught and Honig, 1990a) that the algorithm will compute the relation $\mathbf{U}$ on a CRCW-PRAM with $p(n) = \Theta(n^6)$ processors in $T(n) = O(\log\ n)$ time using $S(n) = O(n^4)$ space.

4 FINAL REMARKS

The slow parallel recognizer is based on a relatively simple idea. In spite of several similarities, it is not a variant of the Cocke-Younger-Kasami (CYK) algorithm or the Earley algorithm (Aho and Ullman, 1972; Harrison, 1978; Earley, 1970); the algebraic definitions specifying the algorithms all differ considerably, and therefore these algorithms all enter their 'items' into their respective matrices for different reasons. Just as for the given algorithm, there exist slow parallel versions of the CYK algorithm and of the Earley algorithm (Nijholt, 1990; Chiang and Fu, 1984).

The topic of fast parallel recognizing and parsing is still young and little research on the subject has been conducted. One of the first publications of a fast parallel recognizer is (Brent and Goldschlager, 1984). Far better known are the results of Gibbons and Rytter. They have described a fast parallel recognizer and parser for grammars in CNF (Gibbons and Rytter, 1988). Unfortunately, CNF is undesirable for many purposes. This is why

we have developed a new fast parallel recognizer that leaves the grammar unchanged. Another recognizer with the same property can be found in (Sikkel and Nijholt, 1991).

Although not given in this paper there also exist parallel parsers which can be used in conjunction with the parallel recognizers. For the slow parallel recognizer there exists a slow parallel parser that can do its job with $\Theta(n)$ processors in $O(n \log\ n)$ time (de Vreught and Honig, 1990b). When the grammar is acyclic, there exists a fast parallel parser running with $\Theta(n^6)$ processors in $O(\log\ n)$ time (de Vreught and Honig, 1990a).

Since the subject of fast parallel parsing is so young, there are many open questions, some of which will probably be solved in the near future. For instance, at this moment it is not yet known whether or not fast parsing of general CFG's is possible without transforming the grammar (we suspect that it is). In addition, determining the behaviour of the algorithms for unambiguous grammars is an interesting research problem.

REFERENCES

Aho, Alfred V. and Ullman, Jeffrey D. 1972 *The Theory of Parsing, Translation and Compiling, Volume I: Parsing*, Prentice Hall, Englewood Cliffs, NJ.

Aho, Alfred V. and Ullman, Jeffrey D. 1973 *The Theory of Parsing, Translation and Compiling, Volume II: Compiling*, Prentice Hall, Englewood Cliffs, NJ.

Akker, Rieks op den; Alblas, Henk; Nijholt, Anton; and Oude Luttighuis, Paul 1989 An Annoted Bibliography on Parallel Parsing, Memoranda Informatica 89-67, University of Twente, Enschede.

Brent, Richard P. and Goldschlager, Leslie M. 1984 A Parallel Algorithm for Context-Free Parsing, *Austral. Comput. Sci. Comm.* 6: 7-1 - 7-10.

Chiang, Y.T. and Fu, King S. 1984 Parallel Parsing Algorithms and VLSI Implementations for Syntactic Pattern Recognition, *IEEE Trans. Pattern Anal. Mach. Intell.* 6: 302-314.

Earley, Jay 1970 An efficient Context-Free Parsing Algorithm, *Commun. ACM* 13(2): 94-102.

Gibbons, Alan and Rytter, Wojciech 1988 *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, MA.

Harrison, Michael A. 1978 *Introduction to Formal Language Theory*, Addison Wesley, Reading, MA.

Nijholt, Anton 1990 The CYK-Approach to Serial and Parallel Parsing, Memoranda Informatica 90-13, University of Twente, Enschede.

Quinn, Michael J. 1987 *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, NY.

Sikkel, Klaas and Nijholt, Anton 1991 An Efficient Connectionist Context-Free Parser, In *2nd Int. Workshop on Parsing Technologies 1991* (these proceedings).

Vreught, Hans de and Honig, Job 1989 A Tabular Bottom Up Recognizer, Reports of the Faculty of Technical Mathematics and Informatics 89-78, Delft University of Technology, Delft.

Vreught, Hans de and Honig, Job 1990a A Fast Parallel Recognizer, Reports of the Faculty of Technical Mathematics and Informatics 90-16, Delft University of Technology, Delft.

Vreught, Hans de and Honig, Job 1990b General Context-Free Parsing, Reports of the Faculty of Technical Mathematics and Informatics 90-31, Delft University of Technology, Delft.

PROCESSING UNKNOWN WORDS IN CONTINUOUS SPEECH RECOGNITION

Kenji Kita, Terumasa Ehara, Tsuyoshi Morimoto

ATR Interpreting Telephony Research Laboratories
Seika-cho, Souraku-gun, Kyoto 619-02, Japan

ABSTRACT

Current continuous speech recognition systems essentially ignore unknown words. Systems are designed to recognize words in the lexicon. However, for using speech recognition systems in real applications of spoken-language processing, it is very important to process unknown words. This paper proposes a continuous speech recognition method which accepts any utterance that might include unknown words. In this method, words not in the lexicon are transcribed as phone sequences, while words in the lexicon are recognized correctly. The HMM-LR speech recognition system, which is an integration of Hidden Markov Models and generalized LR parsing, is used as the baseline system, and enhanced with the trigram model of syllables to take into account the stochastic characteristics of a language. Preliminary results indicate that our approach is very promising.

1 INTRODUCTION

For natural language applications, processing unknown words is one of the most important problems. It is almost impossible to include all words in the system's lexicon.

In the area of written language processing, some methods for handling unknown words have been proposed. For example, Tomita (1986) shows that unknown words can be handled by the generalized LR parsing framework. In generalized LR parsing, it is easy to handle multi-part-of-speech words, and an unknown word can be handled by considering it as a special multi-part-of-speech word.

Unfortunately, in the area of continuous speech recognition, there has been little progress in unknown word processing. Unlike written language processing, in continuous speech recognition, word boundaries are not clear and the correct input is not known, so the problem is more difficult. Recently, Asadi et al. (1990) proposed a method for automatically detecting new words in a speech input. In their method, an explicit model of new words is used to recognize the existence of new words.

This paper proposes a continuous speech recognition method which accepts any utterance that might include unknown words. In our approach, the HMM-LR continuous speech recognition system for Japanese (Kita et al. 1989a; Kita et al. 1989b; Hanazawa et al. 1990) is used as the baseline system, and is an integration of *Hidden Markov Models* (HMM) (Levinson et al. 1983) and *generalized LR parsing* (Tomita 1986). The HMM-LR system is a syntax-directed continuous speech recognition system. The system outputs sentences that the grammar can accept.

The Hidden Markov Model is a stochastic approach for modeling speech, and has been used widely for speech recognition. It is suitable for handling the uncertainty that arises in speech, for example, contextual effects, speaker variabilities, etc. Moreover, if the HMM unit is a phone, then any word models can be composed of phone models. Thus, it is easy to construct a large vocabulary speech recognition system.

In our approach, two kinds of grammars are used. The first grammar is a normal grammar which describes our task. The lexicon for the task is embedded in this grammar as phone

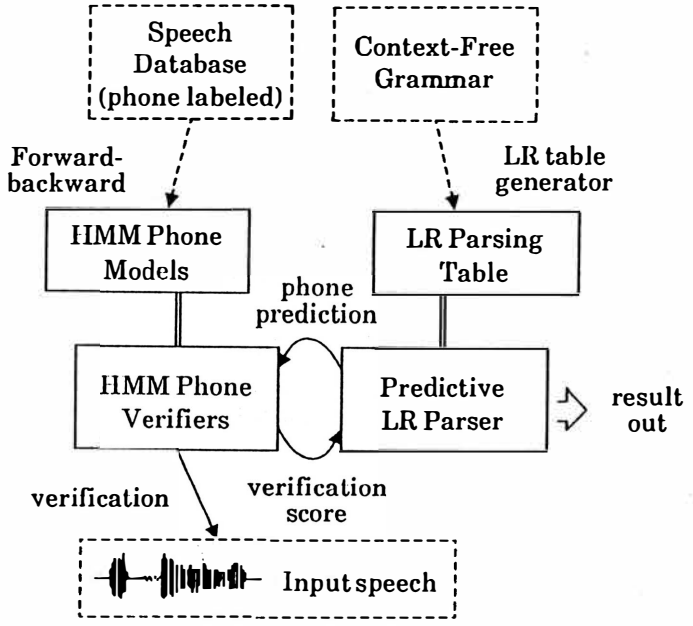


Figure 1: Schematic diagram of HMM-LR speech recognition system

sequences. The second grammar describes the Japanese phonemic structure, in which constraints between phones are written. These two grammars are merged and used in the HMM-LR system. The HMM-LR system outputs words in the lexicon if no unknown word is included in a speech input. If an unknown word is included, then the system outputs a phonemic transcription that corresponds to the unknown word. However, the second grammar by itself is too weak to get correct phonemic transcriptions. We strengthened the grammar by adding other linguistic information, the trigram model based on Japanese syllables. A trigram model is an extremely rough approximation of a language, but it is very practical and useful. By adding the trigram model of syllables, the performance of the system is improved drastically.

2 HMM-LR CONTINUOUS SPEECH RECOGNITION SYSTEM

First, we will review the baseline system, the HMM-LR continuous speech recognition system (Figure 1). This system is an integration of the phone-based HMM and generalized LR parsing.

In HMM-LR, the LR parser is used as a language source model for symbol prediction/generation. Thus, we will hereafter call the LR parser the *predictive LR parser*. A phone-based predictive LR parser predicts next phones at each generation step and generates many possible sentences as phone sequences. The predictive LR parser determines next phones using the LR parsing table of the specified grammar and splits the parsing stack not only for grammatical ambiguity but also for phone variation. Because the predictive LR parser uses context-free rules whose terminal symbols are phone names, the phonetic lexicon for the specified task is embedded in the grammar. An example of context-free gram-

(1)	S	$\rightarrow$	$NP\ VP$
(2)	NP	$\rightarrow$	$DET\ N$
(3)	VP	$\rightarrow$	V
(4)	VP	$\rightarrow$	$V\ NP$
(5)	DET	$\rightarrow$	/z/ /a/
(6)	DET	$\rightarrow$	/z/ /i/
(7)	N	$\rightarrow$	/m/ /ae/ /n/
(8)	N	$\rightarrow$	/ae/ /p/ /a/ /l/
(9)	V	$\rightarrow$	/iy/ /ts/
(10)	V	$\rightarrow$	/s/ /ih/ /ng/ /s/

Figure 2: An example of a grammar with phonetic lexicon

mar rules with a phonetic lexicon is shown in Figure 2. Rule (5) indicates the definite article "the" before consonants, while rule (6) indicates the "the" before vowels. Rules (7), (8), (9) and (10) indicate the words "man", "apple", "eats" and "sings", respectively.

The actual recognition process is as follows. First, the parser picks up all phones predicted by the initial state of the LR parsing table and invokes the HMM models to verify the existence of these predicted phones. The parser then proceeds to the next state in the LR parsing table. During this process, all possible partial parses are constructed in parallel. The HMM phone verifier receives a probability array which includes end point candidates and their probabilities, and updates it using an HMM probability calculation. This probability array is attached to each partial parse. When the highest probability in the array is under a certain threshold level, the partial parse is pruned. The recognition process proceeds in this way until the entire speech input is processed. In this case, if the best probability point reaches the end of the speech data, parsing ends successfully.

High recognition performance is attained by driving HMMs directly without any intervening structures such as a phone lattice. A more detailed algorithm is presented in (Kita et al. 1989a; Kita et al. 1989b).

3 TRIGRAM MODEL OF SYLLABLES

3.1 STOCHASTIC LANGUAGE MODELING

Language models such as context-free grammars or finite state grammars are effective in reducing the search space of a speech recogniton system. These models, however, ignore the stochastic characteristics of a language. By introducing stochastic language models, we can assign the *a priori* probabilities to word/phone sequences. These probabilities, together with acoustic probabilities, determine most likely recognition candidates.

Having observed acoustic data y, a speech recognizer must decide a word sequence $\hat{w}$ that satisfies the following condition:

$$P(\hat{w}|y) = \max_{w} P(w|y)$$

By Bayes' rule,

$$P(w|y) = \frac{P(y|w)P(w)}{P(y)}$$

Since $P(y)$ does not depend on w, maximizing $P(w|y)$ is equivalent to maximizing $P(y|w)P(w)$. $P(w)$ is the *a priori* probability that the word sequence w will be uttered, and is estimated by the *language model*. $P(y|w)$ is estimated by the *acoustic model*. Note that we are using HMM as an acoustic model.

3.2 TRIGRAM MODEL OF SYLLABLES

Word bigram/trigram models are extensively used to correct recognition errors and improve recognition accuracy (Shikano 1987; Paeseler and Ney 1989).

The general idea of a trigram model can be easily applied to Japanese syllables. A typical syllable in Japanese is in the form of a CV, namely one consonant followed by one vowel, and the number of syllables is very small (about one hundred). Moreover, Japanese syllables seem to have a special stochastic structure. Araki et al. (1989) suggest that a statistical method based on Japanese syllable sequences is effective for ambiguity resolution in speech recognition systems. Thus, a syllable trigram model is effective for recognizing Japanese syllable sequences.

In our syllable trigram model, the *a priori* probability $P(S)$ that the syllable sequence $S = s_1, s_2, \ldots, s_n$ will be uttered is calculated as follows (Kita et al. 1990).

$$P(s_1, \ldots, s_n) = P(s_1 \mid \#)P(s_2 \mid \#, s_1) \prod_{k=3}^{n} P(s_k \mid s_{k-2}, s_{k-1}) P(\# \mid s_{n-1}, s_n)$$

$$P(s_k \mid s_{k-2}, s_{k-1}) = q_1 f(s_k \mid s_{k-2}, s_{k-1}) + q_2 f(s_k \mid s_{k-1}) + q_3 f(s_k) + q_4 C$$

$$q_1 + q_2 + q_3 + q_4 = 1$$

$$f(s_k \mid s_{k-2}, s_{k-1}) = \frac{N(s_{k-2}, s_{k-1}, s_k)}{N(s_{k-2}, s_{k-1})}$$

In the above expressions, "#" indicates the phrase boundary marker, and C is a uniform probability that each syllable will occur. The function N counts the number of occurrences of its arguments in the training data. The optimal interpolation weights q_i are determined using *deleted interpolation* (Jelinek and Mercer 1980). Given a collection of training data, the interpolation weights are estimated as follows (Kawabata et al. 1990).

1. Make an initial guess of q_i that $\sum_i q_i = 1$ holds.
2. Calculate i-gram probabilities f_i^j when the j-th data is removed from the training data.
3. Re-estimate q_i by the following formula.

$$\hat{q}_i = \frac{1}{N} \sum_{j=1}^{N} C_i^j$$

where N is the number of syllables in training data, and

$$C_i^j = \frac{q_i f_i^j}{\sum_k q_k f_k^i}$$

4. Replace q_i with $\hat{q}_i$ and repeat from step 2.

4 PROCESSING UNKNOWN WORDS IN AN HMM-LR SPEECH RECOGNITION SYSTEM

4.1 GRAMMAR FOR JAPANESE PHONEMIC STRUCTURE

The HMM-LR system is a syntax-directed continuous speech recognition system. If we use a grammar which describes the Japanese phonemic structure, we can then construct a *phonetic typewriter* for Japanese. This grammar includes rules like "a sequence of consonants doesn't appear" or "the syllabic nasal /N/ doesn't appear at the head of a word". This grammar does not include phonemic spellings for each word, so this grammar is suitable for transcribing an unknown word as a phone sequence.

However, because the *perplexity* [1] of this grammar is quite large, the trigram model of Japanese syllables is used at the same time. By adding the trigram model of syllables, the perplexity of the grammar is reduced from 18.3 to 4.3 (Kawabata et al. 1990).

4.2 UNKNOWN WORD PROCESSING

For processing unknown words, two kinds of grammars are used. The first grammar is a normal grammar which describes our task. The phonemic spellings for each word are also included in this grammar. The second grammar is a grammar for Japanese phonemic structure, mentioned in the previous subsection. Hereafter, these two grammars are referred to as the *task grammar* and the *phonemic grammar*, respectively.

These two grammars are merged and used in the HMM-LR system. When merging two grammars, the start symbol of the phonemic grammar is replaced with pre-terminal names that might include unknown words (in our experiments, *proper-noun* is allowed to include unknown words).

If a speech input includes an unknown word, then a segment of speech input does not match well with any word in the system's lexicon. In this case, the grammar for phonemic structure produces the phone sequence that matches well with the unknown word. If the speech input includes no unknown word, then the HMM-LR system outputs words in the lexicon.

[1] Perplexity is a measurement of language model quality. It represents the average branching of the language model. In general, as perplexity increases, speech recognition accuracy decreases. For more details, see (Jelinek 1990).

4.3 RECOGNITION LIKELIHOOD

The HMM-LR continuous speech recognition system uses the *beam-search* technique to reduce the search space. A group of likely recognition candidates are selected using the likelihood of each candidate. The likelihood S is calculated as follows.

$$S = (1 - \lambda)S^{(HMM)} + \lambda S^{(SYLLABLE)}$$

$S^{(HMM)}$ and $S^{(SYLLABLE)}$ are the log likelihoods based on the HMM and the trigram model of syllables, respectively. The scaling parameter λ is introdued to adjust the scaling of the two kinds of likelihoods, as determined by preliminary experiments.

At the end of recognition, the likelihood of recognition candidates that include unknown words are penalized a small value to reduce the false alarms.

5 EXPERIMENTS

5.1 HMM PHONE MODELS

HMMs used in the experiments are basically the same as reported in (Hanazawa et al. 1990). HMM phone models based on the discrete HMM are used as phone verifiers. A three-loop model for consonants and a one-loop model for vowels are trained using each phone data extracted from the *ATR isolated word database* (Kuwabara et al. 1989).

Duration control techniques and separate vector quantization are used to achieve accurate phone recognition.

5.2 SPEECH DATA

The experiments were carried out using 25 sentences including 279 phrases uttered by one male speaker.

The speech is sampled at 12kHz, pre-emphasized with a filter whose transform function is $(1-0.97z^{-1})$, and windowed using a 256-point Hamming window every 9 msec. Then, 12-order LPC analysis is carried out. Spectrum, difference cepstrum coefficients, and power are computed. Multiple VQ codebooks for each feature were generated using 216 phonetically balanced words. Hard vector quantization without the fuzzy VQ was performed for HMM training. Fuzzy vector quantization (fuzziness = 1.6) was used for test data.

5.3 LINGUISTIC DATA

Syllable trigrams were estimated using a large number of training texts extracted from the *ATR dialogue database* (Ehara et al. 1990). This database contains not only raw texts but also various kinds of syntactic/semantic information, such as *parts of speech*, *pronunciation* and *conjugational patterns*, etc. The training texts includes approximately 73,000 phrases and 300,000 syllables.

5.4 GRAMMARS

As stated earlier, the task grammar and the phonemic grammar are merged into one grammar and used in the HMM-LR system. The task grammar describes the domain of an *International Conference Secretarial Service* and has 1,461 rules including 1,035 words. Of course, all the words which appear in the test data are included in this grammar.

To evaluate the unknown word processing method, all proper nouns (8 words), such as a person's name and a place name, were removed from the task grammar.

5.5 RESULTS

Table 1 shows the transcription rates for phrases that include unknown words. Here the transcription rate is equal to *phone accuracy* (Lee 1989), which can be calculated as follows.

$$phone\,accuracy = \frac{total - sub - ins - del}{total} \times 100$$

where *total* indicates the total number of phones in test data, and *sub*, *ins* and *del* are the number

Table 1: Transcription rates for phrases that include unknown words

Without syllable trigrams	*With syllable trigrams*
66.1%	95.3%

Table 2: Examples of recognition results that include unknown words

Input		*Results*	
Correct	*Meaning*	*Without syllable trigrams*	*With syllable trigrams*
higashiku	higashiku (*place name*)	shigashiku	higashiku
ichitaroudesu	(I am) Ichitarou	ishitaoouutsusu	ishitaroudesu
takarasamadesune	(You are) Mr. Takara (aren't you)	takaasabautsunu	takarasamadesune
kyoutoekikara	from Kyoto station	hyotorekitaafu	kyoutoekikara
kitaoojiekimade	to Kitaooji station	shitaouziekimare	kitaoojiekimade

Table 3: Phrase recognition rates (with syllable trigrams)

rank	*Task grammar*	*Task grammar + Phonemic grammar*
1	87.5%	81.7%
2	93.5%	86.4%
3	94.6%	87.5%

of phones recognized as incorrect, deleted and inserted, respectively.

Table 2 shows examples of recognition results that include unknown words. By using the trigram model of Japanese syllables, the system can output very close phonemic transcriptions for unknown words.

Table 3 shows the phrase recognition rates for two kinds of grammars, the task grammar and a merged grammar consisting of the task grammar and the phonemic grammar. These grammars are both enhanced with the trigram model of syllables. By adding the phonemic grammar, the phrase recognition rate dropped from 87.5% to 81.7%. This is because the phonemic grammar sometimes causes a word to be recognized as a phone sequence despite the word being in the lexicon.

6 CONCLUSION

In this paper, we described a continuous recognition method that can process unknown words. The key idea is merging a task grammar and a phonemic grammar. If no unknown word is included in the speech, then the system uses the task grammar and outputs a correct result. If an unknown word is included, then the system uses the phonemic grammar and outputs a phonemic transcription for the unknown word. We also showed that the trigram model of Japanese syllables is very effective in getting phonemic transcriptions for unknown words.

This is our first approach. There are many problems that must be resolved. Further development to improve the system is currently in progress.

ACKNOWLEDGMENTS

The authors are deeply grateful to Dr. Kurematsu, the president of ATR Interpreting Telephony Research Laboratories, all the members of the *Speech Processing Department* and the *Knowledge and Data Base Department* for their constant help and encouragement.

REFERENCES

[1] Araki, T.; Murakami, J.; and Ikehara, S. 1989 Effect of Reducing Ambiguity of Recognition Candidates in Japanese Bunsetsu Units by 2nd-Order Markov Model of Syllables. *Transactions of Information Processing Society of Japan.* Vol. 30, No. 4 (in Japanese).

[2] Asadi, A.; Schwartz, R. S.; and Makhoul, J. 1990 Automatic Detection of New Words in a Large Vocabulary Continuous Speech Recognition System. *Proceedings of the 1990 International Conference on Acoustics, Speech, and Signal Processing.*

[3] Ehara, T.; Ogura, K.; and Morimoto, T. 1990 ATR Dialogue Database. *Proceedings of the International Conference on Spoken Language Processing.*

[4] Hanazawa, T.; Kita, K.; Nakamura, S.; Kawabata, T.; and Shikano, K. 1990 ATR HMM-LR Continuous Speech Recognition System. *Proceedings of the 1990 International Conference on Acoustics, Speech, and Signal Processing.* Also In: Waibel, A. and Lee, K. F. (eds.) *Readings in Speech Recognition.* Morgan Kaufmann Publishers.

[5] Jelinek, F. and Mercer, R. L. 1980 Interpolated Estimation of Markov Source Parameters from Sparse Data. In: Gelsema, E. S. and Kanal, L. N. (eds.) *Pattern Recognition in Practice.* North Holland.

[6] Jelinek, F. 1990 Self-Organized Language Modeling for Speech Recognition, In: Waibel, A. and Lee, K. F. (eds.) *Readings in Speech Recognition.* Morgan Kaufmann Publishers.

[7] Kawabata, T.; Hanazawa, T.; Itoh, K.; and Shikano, K. 1990 HMM Phone Recognition Using Syllable Trigrams. *IEICE Technical Report.* SP89-110 (in Japanese).

[8] Kita, K.; Kawabata, T.; and Saito, H. 1989a HMM Continuous Speech Recognition Using Predictive LR Parsing. *Proceedings of the 1989 International Conference on Acoustics, Speech, and Signal Processing.*

[9] Kita, K.; Kawabata, T.; and Saito, H. 1989b Parsing Continuous Speech by HMM-LR Method. *First International Workshop on Parsing Technologies.*

[10] Kita, K.; Kawabata, T.; and Hanazawa, T. 1990 HMM Continuous Speech Recognition Using Stochastic Language Models. *Proceedings of the 1990 International Conference on Acoustics, Speech, and Signal Processing.*

[11] Kuwabara, H.; Takeda, K.; Sagisaka, Y.; Katagiri, S.; Morikawa, S.; and Watanabe, T. 1989 Construction of a Large-Scale Japanese Speech Database and its Management System. *Proceedings of the 1989 International Conference on Acoustics, Speech, and Signal Processing.*

[12] Lee, K. F. 1989 *Automatic Speech Recognition: The Development of the SPHINX System.* Kluwer Academic Publishers.

[13] Levinson, S. E.; Rabiner, L. R.; and Sondhi, M. M. 1983 An Introduction to the Application of the Theory of Probabilistic Functions of a Markov Process to Automatic Speech Recognition. *Bell System Technical Journal.* Vol. 62, No. 4.

[14] Paeseler, A. and Ney, H. 1989 Continuous-Speech Recognition Using a Stochastic Language Model. *Proceedings of the 1989 International Conference on Acoustics, Speech, and Signal Processing.*

[15] Shikano, K. 1987 Improvement of Word Recognition Results by Trigram Model. *Proceedings of the 1987 International Conference on Acoustics, Speech, and Signal Processing.*

[16] Tomita, M. 1986 *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers.

February 14, 1991

Session C

The Specification and Implementation of Constraint-Based Unification Grammars*

Bob Carpenter Carl Pollard† Alex Franz

Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213
(412) 268-8573 carp@lcl.cmu.edu
†Linguistics Department, Ohio State University

Summary

Our aim is to motivate and provide a specification for a unification-based natural language processing system where grammars are expressed in terms of principles which constrain linguistic representations. Using typed feature structures with multiple inheritance for our linguistic representations and definite attribute-value logic clauses to express constraints, we will develop the bare essentials required for an implementation of a parser and generator for the Head-driven Phrase Structure Grammar (HPSG) formalism of Pollard and Sag (1987).

1 Introduction

In the past decade, two competing approaches to the scientific study of natural language grammar have become predominant, the *rule-based* approach and the *principle/constraint-based* approach. Within the rule-based approach, exemplified by Lexical Functional Grammar (LFG) (Bresnan 1982) and Generalized Phrase Structure Grammar (GPSG) (Gazdar *et al.* 1985), rules are taken to correspond to grammatical *constructions* and are modeled as more or less schematic productions with the well-formed structures of the language generated over a finite set of *lexical items* by recursively applying the rules. Both LFG and GPSG are based upon context-free skeletons and explain syntactic dependencies in terms of informational consistency constraints that can be solved using feature structure unification. There has been a great deal of success in implementing these formalisms, in part due to their declarative nature and natural semantics, but also due to the existence of general unification-based grammar processing systems such as Functional Unification Grammar (FUG) (Kay 1985) and PATR-II (Shieber *et al.* 1983).

Principle-based approaches to grammar have become predominant in theoretical linguistics, primarily due to the influence of Chomsky's (1981) Government-Binding (GB) framework. The novel aspect of GB considered as a grammar formalism is that it advocates the total abandonment of construction-specific rules in favor of a collection of interacting principles which serve to delimit the well-formed linguistic structures. Candidate structures are generated according to extremely general, universal, phrasal immediate dominance (ID) schemata ($\bar{X}$ Theory) and then iteratively transformed using movement rules (Move-α) in accordance with a number of highly tuned principles to deal with case (Case Theory), complementation (Projection Principle), pronominal and other coreference (Binding Theory), long-distance dependencies

*The authors would like to thank Bob Kasper for a number of useful comments on an earlier draft of this paper. The research of Pollard and Franz was supported by a grant from the National Science Foundation (IRI-8806913).

(Empty Category Principle and Subjacency) and so forth. Patterns of cross-linguistic variation are accounted for by means of the *parametrization* of these principles.

The methodological distinction between these two approaches is widely supposed to be that rules enumerate possibilities, while principles eliminate possibilities. But it is quite difficult to distinguish formally between a parametrized disjunctive principle and a collection of schematic rules only one of which can apply to a given structure. Consider, for example, the distinction between categorial grammar application schemata, basic ID rules of GPSG, and the C-structure constraints of LFG, on the one hand, and the disjunctive clauses of $\bar{X}$ Theory or the Empty Category Principle on the other. It should also be borne in mind that so-called rule-based approaches often employ not only rules but also global constraints on representations which behave similarly to principles, such as the Head Feature Convention and the Control Agreement Principle of GPSG or the Completeness and Function-Argument Biuniqueness Conditions of LFG.

HPSG belongs to the "unification-based" family of linguistic theories, but differs from LFG and GPSG in that grammars are formulated entirely in terms of universal and language-specific principles expressed as constraints on feature structures, which in turn are taken to represent possible linguistic objects. As shown by Pollard and Sag (1987), constraints on feature structures can be used to do the same duty as many of the principles and rules of GPSG, LFG and GB. Unlike rule-based theories, in HPSG, immediate dominance and linear precedence conditions (traditional phrase-structure) are not modeled any differently than other constraints. But like the rule-based approaches, there is no appeal to derivational notions such as movement; the work of transformations in GB is taken over by declarative constraints stated at a single level of representation.

Departing from more traditional formalisms which employ phrase-structure trees as the primary device for linguistic representation, we follow HPSG (and to some extent LFG) in representing linguistic objects as feature structures. To this end, we show how a natural type discipline can be imposed on feature structures allowing for multiple inheritance and the specification of feature appropriateness and value restrictions. Our typing will be strong in the sense that every feature structure must be associated with a type. Strong typing carries with it the usual benefits of early error detection and enhanced control over crucial memory allocation, access and reclamation functions. The use of multiple inheritance allows a sophisticated network of constraints to be expressed at the appropriate level of detail. This is especially important for the development of large lexicons (Flickinger *et al.* 1985).

Our types can be used to represent information that must be encoded by expensive structural unification or inference steps in untyped systems. In automatic deduction systems, this has been found to provide a significant run-time gain due to the fact that useless branches in the search space can be efficiently detected and pruned before the creation of expensive structural copies or binding frames (Walther 1985, 1988; Aït-Kaci and Nasr 1986).

In a constraint-based linguistic theory such as HPSG, parsing and generation reduces to solving constraints. We allow constraints to be expressed by a feature logic analogue of definite clauses. The benefit of this approach is that it admits a natural and effective method paralleling SLD-resolution (see Lloyd 1984) for enumerating the solutions to a system of constraints.

2 Inheritance and Appropriateness

Type declarations in our system contain information concerning subtyping and *appropriateness conditions* which state the features that are appropriate for each type and the values that they can take.

Definition 1 (Type Scheme) *A* type scheme *is a tuple* $\Sigma = \langle \mathsf{Type}, \sqsubseteq, \mathsf{Feat}, Approp \rangle$ *where*

- $\langle \mathsf{Type}, \sqsubseteq \rangle$ *is a finite consistently complete*[‡] *partial order of the* types *by* subsumption, *called the* inheritance hierarchy
- Feat *is a finite set of* features
- $Approp : \mathsf{Feat} \times \mathsf{Type} \rightarrow \mathsf{Type}$ *is a partial function such that:*
  - (Minimal Introduction)
    for every f *there is a least type* σ *such that* $Approp(f, \sigma)$ *is defined*
  - (Upward Closure and Monotonicity)
    if $\sigma \sqsubseteq \tau$ *and* $Approp(f, \sigma)$ *is defined then* $Approp(f, \tau)$ *is defined and* $Approp(f, \sigma) \sqsubseteq Approp(f, \tau)$

If $\sigma \sqsubseteq \tau$ we say that σ *subsumes*, is *more general* than or a *supertype* of τ. We refer to the least upper bound operation in our inheritance hierarchy as *(type) unification* since the least upper bound of a set of objects representing partial information is the object which represents the most general piece of information that is more specific than each member of the set. The least upper bound of the empty set is written $\bot$, read "bottom", and is the unique *universal* or most general type. Our restrictions on appropriateness are analogous to the condition of regularity in the signatures of order-sorted algebras (Meseguer *et al.* 1987); taken together, the conditions on the inheritance hierarchy and appropriateness function will ensure that unification is well-defined and produces a unique result, which is crucial for efficient and natural unification-based processing (Pereira 1987).

3 Feature Structures

We will begin by introducing an untyped collection of *feature structures* which are similar to the ψ-terms of Aït-Kaci (1984) and the sorted feature structures of Smolka (1988) and Pollard and Moshier (1990).

Definition 2 (Feature Structure) *A* feature structure *is a tuple* $F = \langle Q, \bar{q}, \theta, \delta \rangle$ *where*

- $\bar{q}$ *: the* root node *in* Q
- Q *: a finite set of* nodes *rooted at* $\bar{q}$ *so that* $Q = \{\delta(\pi, \bar{q}) \mid \pi \in \mathsf{Path}\}$ *(see below for definition of* $\delta(\pi, q)$ *and* Path*)*

[‡]A subset X of a partial ordering $\langle S, \sqsubseteq \rangle$ is said to be ***consistent*** if it has an upper bound. A partial order is ***consistently complete*** if every (possibly empty) consistent set X has a ***least*** upper bound, which we write $\bigsqcup X$, or $x \sqcup y$ when $X = \{x, y\}$.

- $\theta : Q \rightarrow \mathsf{Type}$ *: a total node* type assignment
- $\delta : \mathsf{Feat} \times Q \rightarrow Q$ *: a partial* feature value function

Thus a feature structure is a rooted, connected, directed graph with vertices labeled by types and edges labeled by features. We will write $q : \sigma \xrightarrow{f} q' : \sigma'$ if $\delta(f, q) = q'$ and $\theta(q) = \sigma$ and $\theta(q') = \sigma'$. We think of each node as representing a partial frame or record with values for its slots given by its outgoing arcs.

We let $\mathsf{Path} = \mathsf{Feat}^*$ be the set of *paths*, which consist of finite sequences of features. We let ϵ denote the empty path and extend δ to paths by setting $\delta(\epsilon, q) = q$ and $\delta(f\pi, q) = \delta(\pi, \delta(f, q))$. Our definition requires that every node be reachable from the root node $\bar{q}$, where $\delta(\pi, \bar{q})$ is the node that can be reached from $\bar{q}$ along the path π.

Note that we have not disallowed *cyclic* feature structures, in which there is some non-empty path π and node q such that $\delta(\pi, q) = q$.

We extend our ordering on types to an ordering of the feature structures in the usual way (see Pollard and Moshier 1990).

Definition 3 (Subsumption) $F = \langle Q, \bar{q}, \theta, \delta \rangle$ subsumes $F' = \langle Q', \bar{q}', \theta', \delta' \rangle$, $F \sqsubseteq F'$, *iff there is a total* $h : Q \rightarrow Q'$ *such that*

- $h(\bar{q}) = \bar{q}'$
- $\theta(q) \sqsubseteq \theta'(h(q))$ *for every* $q \in Q$
- $h(\delta(f, q)) = \delta'(f, h(q))$ *for every* $q \in Q$ *and feature* f *such that* $\delta(f, q)$ *is defined*

The last two conditions on h can be stated graphically as requiring that if $q : \sigma \xrightarrow{f} q' : \sigma'$ in F then $h(q) : \tau \xrightarrow{f} h(q') : \tau'$ in F' and furthermore, $\sigma \sqsubseteq \tau$ and $\sigma' \sqsubseteq \tau'$. Such a mapping takes each node of the more general structure onto a node of the more specific structure in a way that preserves structure sharing and does not lose any type information.

Subsumption is only a *pre-ordering*, so we write $F \sim F'$ if $F \sqsubseteq F'$ and $F' \sqsubseteq F$ and say that F and F' are *alphabetic variants*. We could work in the collection of feature structures modulo alphabetic variance, which is guaranteed to be a partial order, but this becomes tedious (for an elegant approach to representing these equivalence classes, see Moshier (1988)). In our situation, with only a pre-order, we define a *unifier* of a pair of feature structures F and F' to be any feature structure F'' such that $F \sqsubseteq G$ and $F' \sqsubseteq G$ if and only if $F'' \sqsubseteq G$. The primary result concerning subsumption is stated as follows:

Theorem 4 (Unification) *Unique unifiers exist for pairs of consistent feature structures, up to alphabetic variance.*

Proof: The usual unification algorithm for feature structure works with the addition of a step that unifies the types of the inputs to produce the type of the result and fails if the types are not consistent. See Aït-Kaci (1984) or Pollard and Moshier (1990). □

In theory, the asymptotic behavior of the unification algorithm is not affected; type unification can be carried out by table look-up. In practice, the negligible constant overhead of type unification at every step of the process will actually save time in that inconsistencies can be detected before any recursive structures need to be inspected.

We now define a notion of typing which singles out some of the feature structures as being well-typed. Intuitively, a feature structure is well-typed if every feature that appears is appropriate and takes an appropriate value.

Definition 5 (Well-Typing) *A feature structure* $F = \langle Q, \bar{q}, \theta, \delta \rangle$ *is* well-typed *if* $q : \sigma \xrightarrow{f} q' : \sigma'$ *in* F *implies* $Approp(f, \sigma) \sqsubseteq \sigma'$.

If F is a feature structure and F' a well-typed feature structure such that $F \sqsubseteq F'$ then we say that F' is a *well-typed extension* of F and that F is *typable*.

Fortunately, the user does not need to specify all of the values for appropriate features about which nothing is known; a type inference procedure can be defined that determines the minimum possible types that will extend a feature structure so that it is well-typed.

Theorem 6 (Type Inference) *There is an effectively computable partial function* $TypInf$ *from the feature structures onto the well-typed feature structures such that* $TypInf(F)$ *is defined if and only if* F *is typable. In that case* $F \sqsubseteq F'$ *for a well-typed* F' *if and only if* $TypInf(F) \sqsubseteq F'$.

Proof: A constructive type inference procedure can proceed by successively increasing the types on those nodes which do not yet meet the appropriateness conditions. All that is required is the iteration of the following steps until a closure is reached:

- if a feature is defined at a node, the type of the node should be unified with the minimal type appropriate for the feature.
- if a feature is defined and its value is not of great enough type, unify in the type for the minimal value.

Thus every typable feature structure has a minimal well-typed extension which is unique up to alphabetic variance. This process is not sensitive to the order in which nodes and features are chosen. It is also guaranteed to terminate as there are only a finite number of types and nodes to start with. To see that the result is minimal, simply notice that each operation in the iteration was required so that the result is well-typed. □

The function $TypInf$ displays a host of interesting properties. For instance, it can be factored into two separate operations corresponding to the two steps in $TypInf$. It is not hard to see that that $F \sqsubseteq TypInf(F)$, $TypInf(F) = TypInf(TypInf(F))$, and $F \sqsubseteq F'$ implies that $TypInf(F) \sqsubseteq TypInf(F')$. More significantly, we have:

(1) $TypInf(TypInf(F_1) \sqcup \cdots \sqcup TypInf(F_n)) = TypInf(F_1 \sqcup \cdots \sqcup F_n)$

whenever the latter exists. This means that we can be as lazy as we like about type inference at run time without fear of information loss. It also follows that the type inference procedure can be composed with a unification procedure for feature structures to provide a unification procedure for well-typed feature structures.

Theorem 7 (Well-Typed Unification) *If* F *and* F' *are consistent well-typed feature structures such that* $F \sqcup F'$ *is typable, then* $TypInf(F \sqcup F')$ *is their least upper bound in the collection of well-typed feature structures (modulo alphabetic variance).*

The significance of this theorem is that it will be possible to compute the least specific well-typed feature structure that extends a consistent pair of well-typed feature structures.

This notion of well-typing is not the only one possible. It is also sensible to consider a stronger notion of typing whereby every feature that is appropriate must be defined. This notion, called *total* well-typing, corresponds to the composition of $TypInf$ with a second closure operator that adds in features that have not been defined in $TypInf(F)$ and gives them their minimal values. As Franz (1990) points out, the appropriateness conditions must meet a certain acyclicity condition to ensure the termination of type inference for this stronger notion of typing. These more strongly typed systems allow better management of memory since feature structures of a given type are of a known size and can have their feature values indexed positionally rather than by feature/value pairs. On the other hand, the notion of well-typing that we consider here is simpler and is also better suited to applications in which the number of features containing information is sparse relative to the number of possible features that can be defined for any given feature structure. For instance, in the application to HPSG we provide below, feature structures occurring early in the search space will be quite sparse compared to their later instantiations.

4 Feature Logic

We can describe our feature structures with a variant of the feature logic introduced by Rounds and Kasper (1986). We present a simultaneous definition of both the *well-formed formulas* or *descriptions* and of *satisfaction* of a formula by a feature structure, which we write $F \models \phi$:

Definition 8 (Formulas and Satisfaction)

Formula	Satisfaction Condition
$F \models \sigma$	*the root node of* F *is assigned a type at least as specific as* σ
$F \models \pi : \phi$	*the value of* F *at* π *is defined and satisfies* ϕ
$F \models \pi \doteq \pi'$	*the paths* π *and* π' *lead to the same node in* F
$F \models \phi \wedge \psi$	$F \models \phi$ *and* $F \models \psi$.
$F \models \phi \vee \psi$	$F \models \phi$ *or* $F \models \psi$.

The behavior of this logic on the typed feature structures we present here can be given a complete equational axiomatization along the lines of Rounds and Kasper (1986) by adding in additional axioms for type unification (Pollard in press) and appropriateness (Pollard and Carpenter to appear). The primary result of Rounds and Kasper carries over to the present situation:

Theorem 9 (Minimal Satisfiers) *For every formula* ϕ *there is a finite set* $\{F_0, \ldots, F_{n-1}\}$ *of pairwise incomparable feature structures, unique up to alphabetic invariance, such that* $F \models \phi$ *if and only if* $F_i \sqsubseteq F$ *for some* $i < n$.

Proof: The proof of Rounds and Kasper (1986) can be easily adapted by applying the type inference procedure. The key result is that the set of minimal satisfiers of a conjunction is derived by the pairwise unification of the minimal satisfiers of the conjuncts. □

5 Constraint Systems and Solutions

Departing from Pollard and Sag (1987) and following Pollard and Moshier (1990), we attach constraints to types rather than allowing general implicative and negative constraints. The

constraints attached to types will be much more expressive than the easily decidable conditions arising from the inheritance and appropriateness conditions in the type scheme which are only intended to specify the class of well-typed feature structures over which the constraints range.

Definition 10 (Constraint System) *A* constraint system Φ *associates each type* τ *with a feature logic formula* Φ_τ.

We provide for the multiple inheritance of constraints, letting $\downarrow\Phi_\tau$ be the conjunction of the constraints associated with τ and all of its supertypes; formally $\downarrow\Phi_\tau = \bigwedge_{\sigma \sqsubseteq \tau} \Phi_\sigma$. Since the feature structures in Φ_σ may contain arbitrary types, the system Φ may be recursive. Pollard and Sag (1987) show how systems of constraints of this general form can be used to model not only language-specific grammars, but also entire linguistic theories (universal grammars).

In general, we will be interested in solving *queries* with respect to systems of constraints, where a query simply consists of an feature logic description. In applications to parsing, a query would represent the value of the phonology feature and a constraint on the syntactic category of the result; for generation, a query might represent instantiated semantic and pragmatic features. A solution is then a well-typed feature structure which satisfies both the query and all of the constraints expressed by the grammar.

Definition 11 (Solution) *A feature structure* F *is a solution to a query* ψ *with respect to a system* Φ *of constraints just in case* $F \models \psi$ *and the maximal substructure* F_q *rooted at each node* q *of* F *satisfies the inherited constraint on its type* $\theta(q)$, *so that* $F_q \models \downarrow\Phi_{\theta(q)}$.

We will provide a complete method for generating the solutions to queries with respect to constraint systems that is defined in terms of non-deterministic feature structure rewriting. Our method is inspired by the rewriting operation employed by Aït-Kaci (1984), which, to the best of our knowledge, was the first programming system based upon recursively defined constraints on feature structures; but our method is cleaner in that it provides a strong distinction between the logical language and its feature structure models and also more general in that it applies to cyclic feature structures. More importantly, our system is provably complete.

The basic operation of rewriting is to non-deterministically choose a node in the feature structure and then non-deterministically choose a minimal satisfier for the inherited constraint associated with the type attached to that node to be unified into the feature structure. This is analogous to SLD-resolution as applied to definite clauses, in which a subgoal is replaced by the body of a clause after unifying the head of the clause with the subgoal.

Let $\pi \cdot F$ be the feature structure consisting of the path π with F attached to its terminal node.

Definition 12 (Rewriting) *If* F *is a well-typed feature structure where the node at path* π *is assigned type* σ *and if* G *is a minimal satisfier of the inherited constraint* $\downarrow\Phi_\sigma$ *on* σ *then* rewriting *is defined so that* $F \stackrel{\pi}{\Longrightarrow} TypInf(F \sqcup \pi \cdot G)$.

Of course, as we mentioned earlier, type inference can be interleaved arbitrarily with this rewriting operation. We can also interleave rewriting along arbitrary paths, using the notation $F \Longrightarrow F'$ if $F \stackrel{\pi}{\Longrightarrow} F'$ for some path π.

Our next theorem shows that minimal solutions can be effectively generated by rewriting. In effect, it is the completeness theorem for our operational interpretation; it tells us that every solution can be found by rewriting. In particular, a breadth-first enumeration of the search space determined by the rewriting system will eventually uncover every solution.

Theorem 13 (Solution) *F is a* solution *to the query ψ with respect to the constraint system Φ if and only if $F \stackrel{\pi}{\Longrightarrow} F$ for every path π for which F is defined. F is a minimal solution if and only if there is a derivation of F by rewriting from a minimal satisfier of ψ.*

Proof: (Sketch) The conditions on a solution are just that every node satisfy the constraint on its type. This happens if and only if the unifying in of a minimal satisfier to the constraint does not add any new information.

The usual fixed-point style induction suffices to establish minimality. Suppose we fix a solution to the query ψ. In the base case, this solution must be more specific than a minimal satisfier of the query ψ. The inductive hypothesis is that at every stage during rewriting we are dealing with a feature structure which subsumes the solution.

During rewriting, we unify in constraints associated with more general types than in the solution since we have a feature structure which subsumes the solution. Since we inherit constraints, rewriting can be done so that it unifies in a minimal satisfier to a constraint which subsumes the minimal satisfier associated with the corresponding node in the solution. The rewriting process will eventually reach a solution after a finite number of steps or continue on indefinitely, because there are only a finite number of steps that can be taken without adding in more nodes due to the finite number of nodes in a feature structure and finite number of types in the inheritance hierarchy.

If rewriting reaches a solution, then by the inductive hypothesis, that solution must be at least as general as the given solution. Finite satisfiers which are not generated by rewriting from a minimal satisfier of the query can thus not be minimal. □

For the sake of brevity we have not discussed constraints which express n-ary relational dependencies between path values. An example of a relational dependency expression would be $append(\pi_1, \pi_2, \pi_3)$, where π_1, π_2, and π_3 are paths; this means that the value of the path π_3 must be the concatenation of the values of π_1 and π_2. Such relations can be given definite-clause-style recursive definitions, as in:

$$(2) \quad \mathbf{append}(\pi_1, \pi_2, \pi_3) \leftarrow (\pi_1 : \mathbf{nil} \wedge \pi_2 \doteq \pi_3) \vee (\pi_1 \cdot \textsc{first} \doteq \pi_3 \cdot \textsc{first} \wedge \mathbf{append}(\pi_1 \cdot \textsc{rest}, \pi_2, \pi_3 \cdot \textsc{rest}))$$

Adding definitions of this kind to our feature logic is somewhat analogous to augmenting an underlying constraint language with definite relations as proposed by Höhfeld and Smolka (1988). However, it should be borne in mind that the π_i in our definition schemata are path parameters, not genuine logical variables. Aït-Kaci (1984) showed how relations could be encoded as types with arguments specified by features and arbitrary constraints for definitions; each use of such a relation then requires a node in a feature structure at which to be anchored (usually as the value of a so-called *garbage feature*). Franz (1990) implemented relations directly, requiring their arguments to be typed; in the case of *append*, all of the arguments would be of type **list**, which has two subtypes: **nil** (empty list), which is not appropriate for any features, and **ne-list** (nonempty list), which is appropriate for the features FIRST with value restriction ⊥ and REST with value restriction **list**.

6 Implementation

The typed system described here has been implemented in both Lisp (Franz 1990) and Prolog. Emele and Zajac (personal communication) report that Franz's (1990) grammar has been

ported, with a 100-fold speedup, to their TFS system (1990) which was originally based on Aït-Kaci (1984, 1986). We anticipate that a number of the optimizations employed in TFS will carry over to the system described here. In Franz's system, compilation is first carried out on the type scheme and constraints to detect errors and compute minimal satisfiers. A serious processing bottleneck can be traced to the search incurred by disjunctive constraint solving. This naturally leads to the issue of which search strategy should be employed. The conclusion of Franz (1990) was that specialized search strategies would be needed for linguistic applications. Ideally, a general mechanism for specifying search preference would be provided.

The complexity of the basic operations of this system is very low. Subsumption can be computed in linear time by explicit construction of the mapping function. Similarly, efficient near-linear unification algorithms can be used (Jaffar 1984). On the other hand, disjunctive representations are very compact in that the number of minimal satisfiers for a formula is exponential in the size of the formula in the worst case and satisfiability of a formula is $\mathcal{NP}$-complete (Kasper and Rounds 1986). Relatively efficient practical algorithms for dealing with disjunctions have been developed by Kasper (1987) and Eisele and Dörre (1988). Another option that is being explored is the utilization of total typing as discussed above, for managing memory allocation and improving the speed of both unification and the unwinding of information upon backtracking. The features values themselves could then be retrieved automatically without searching through a collection of feature-value pairs. Hopefully, compilation and run-time optimization techniques employed for logic programs can also be directly incorporated, such as type indexing for rules and deterministic tree pruning.

Furthermore, the connections between constraint-based grammars and terminological knowledge representations based on inheritance networks such as KL-ONE (Brachman and Schmolze 1985) and especially its descendant LOOM (Mac Gregor 1988) has only begun to be explored (Kasper 1989, Nebel and Smolka 1989); there is a great deal of promise that insights from these systems can be employed to produce more powerful and efficient type inference and search techniques. Kasper and Pollard are currently exploring the possibility of a chart-parser analog for HPSG-style grammars that exploits the possible-worlds mechanism of LOOM for conceptually clean and space-efficient structure sharing within the chart.

There are many possible extensions that could be added to our constraint systems. In particular, Pollard and Moshier (1990) have provided a compatible account of set valued feature structures, Carpenter (1990) has added a notion of inequation analagous to the inequations of Prolog II (Colmerauer 1984), and a general notion of feature structure extensionality is discussed in Pollard and Carpenter (to appear).

One thing that this system shares with PATR-II and other general unification-based systems is that while the solutions to queries can be recursively enumerated, it is undecidable whether a query has a solution. While we do not present a proof here, the result follows from the fact that logic programs and queries can be reduced to the solution of a system of constraints (the trick is to include proof trees as a type and encode the notion of an acceptable proof tree with respect to a program as a constraint on its type). Of course, this does not render our system unusable any more than Prolog or PATR-II are rendered useless by their undecidability; it just means that the user must exercise due caution in constructing linguistically reasonable grammars, in order to ensure that all-paths parsing always terminates. In generation, of course, nontermination is to be expected; but in this case, fortunately, a single solution will suffice.

References

Aït-Kaci, H. (1984). *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially Ordered Types.* PhD thesis, University of Pennsylvania.

Aït-Kaci, H. (1986). An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351.

Aït-Kaci, H. and Nasr, R. (1986). Login: A logical programming language with built-in inheritance. *Journal of Logic Programming*, 3:187–215.

Brachman, R. J. and Schmolze, J. G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216.

Bresnan, J. W., editor (1982). *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Massachusetts.

Carpenter, B. (1990). Typed feature structures: Inheritance, (in)equations and extensionality. In *Proceedings of the First International Workshop on Inheritance and Natural Language*, Tilburg, The Netherlands.

Chomsky, N. (1981). *Lectures on Government and Binding.* Foris, Dordrecht.

Colmerauer, A. (1984). Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo.

Eisele, A. and Dörre, J. (1988). Unification of disjunctive feature descriptions. In *Proceedings of the 26th Annual Conference of the Association for Computational Linguistics*, Buffalo, New York.

Emele, M. C. and Zajac, R. (1990). Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*, Helsinki, Finland.

Flickinger, D., Pollard, C. J., and Wasow, T. (1985). Structure-sharing in lexical representation. In *Proceedings of the 23rd Annual Conference of the Association for Computational Linguistics.*

Franz, A. (1990). A parser for HPSG. Technical Report LCL-90-3, Laboratory for Computational Linguistics, Carnegie Mellon University, Pittsburgh.

Gazdar, G., Klein, E., Pullum, G., and Sag, I. (1985). *Generalized Phrase Structure Grammar.* Basil Blackwell, Oxford.

Höhfeld, M., and Smolka, G. (1988) Definite Relations over Constraint Languages. LILOG-REPORT 53, IBM Deutschland GmbH, Stuttgart, FRG.

Jaffar, J. (1984). Efficient unification over infinite terms. *New Generation Computing*, 2:207–219.

Johnson, M. (1988). *Attribute-Value Logic and the Theory of Grammar*, volume 14 of *Lecture Notes.* Center for the Study of Language and Information, Stanford, California.

Kasper, R. T. (1987). A unification method for disjunctive feature structures. In *Proceedings of the 25th Annual Conference of the Association for Computational Linguistics*, pages 235–242.

Kasper, R. T. (1989). Unification and classification: An experiment in information-based parsing. In *First International Workshop on Parsing Technologies*, pages 1–7, Pittsburgh.

Kasper, R. T. and Rounds, W. C. (1986). A logical semantics for feature structures. In *Proceedings of the 24th Annual Conference of the Association for Computational Linguistics*, pages 235–242.

Kay, M. (1985). Parsing in functional unification grammar. In Dowty, D. R., Karttunen, L., and Zwicky, A., editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press, London.

King, P. (1989). *A Logical Formalism for Head-Driven Phrase Structure Grammar.* PhD thesis, University of Manchester, Manchester, England.

Lloyd, J. W. (1984). *Foundations of Logic Programming.* Springer-Verlag, West Berlin, FRG.

Mac Gregor, R. (1988). A deductive pattern matcher. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, pages 403–408, St. Paul, Minnesota.

Meseguer, J., Goguen, J., and Smolka, G. (1987). Order-sorted unification. Technical Report CSLI-87-86, Center for the Study of Language and Information, Stanford University, Stanford, California.

Moshier, D. (1988). *Extensions to Unification Grammar for the Description of Programming Languages.* PhD thesis, University of Michigan, Ann Arbor.

Moshier, M. A. (1989). A careful look at the unification algorithm. Unpublished Manuscript, Department of Mathematics, University of California, Los Angeles.

Mycroft, A. and O'Keefe, R. A. (1984). A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307.

Nebel, B. and Smolka, G. (1989). Representation and reasoning with attributive descriptions. IWBS Report 81, IBM – Deutschland GmbH, Stuttgart, FRG.

Pereira, F. C. (1987). Grammars and logics of partial information. In Lassez, J.-L., editor, *Proceedings of the Fourth International Symposium on Logic Programming*, pages 989–1013.

Pereira, F. C. N. and Shieber, S. M. (1984). The semantics of grammar formalisms seen as computer languages. In *Proceedings of the 10th International Conference on Computational Linguistics*, pages 123–129.

Pollard, C. J. (in press). Sorts in unification-based grammar and what they mean. In Pinkal, M. and Gregor, B., editors, *Unification in Natural Language Analysis.* MIT Press.

Pollard, C. J. and Carpenter, B. (to appear). Extensionality in Feature Structures and Feature Logic. Paper presented at the Workshop on Unification and Generation, Bad Teinach, Germany, November 1990. To appear in the Proceedings.

Pollard, C. J. and Moshier, M. D. (1990). Unifying partial descriptions of sets. In Hanson, P., editor, *Information, Language and Cognition*, volume 1 of *Vancouver Studies in Cognitive Science.* University of British Columbia Press, Vancouver.

Pollard, C. J. and Sag, I. A. (1987). *Information-Based Syntax and Semantics: Volume I – Fundamentals*, volume 13 of *CSLI Lecture Notes.* Chicago University Press, Chicago.

Rounds, W. C. and Kasper, R. T. (1986). A complete logical calculus for record structures representing linguistic information. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, Cambridge, Massachusetts.

Shieber, S. M. (1986). *An Introduction to Unification-Based Approaches to Grammar*, volume 4 of *CSLI Lecture Notes.* Chicago University Press, Chicago.

Shieber, S. M., Uszkoreit, H., Pereira, F. C. N., Robinson, J., and Tyson, M. (1983). The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*, volume 1894 of *SRI Final Report.* SRI International, Menlo Park, California.

Smolka, G. (1988). A feature logic with subsorts. LILOG-REPORT 33, IBM Deutschland GmbH, Stuttgart, FRG.

Walther, C. (1985). A mechanical solution of Schubert's Steamroller by many-sorted resolution. *Artificial Intelligence*, 26(2):217–224.

Walther, C. (1988). Many-sorted unification. *Journal of the ACM*, 35:1–17.

PROBABILISTIC LR PARSING FOR GENERAL CONTEXT-FREE GRAMMARS*

See-Kiong Ng and Masaru Tomita
School of Computer Science and Center for Machine Translation
Carnegie Mellon University
Pittsburgh, PA 15213 U.S.A.

ABSTRACT

To combine the advantages of probabilistic grammars and generalized LR parsing, an algorithm for constructing a probabilistic LR parser given a probabilistic context-free grammar is needed. In this paper, implementation issues in adapting Tomita's generalized LR parser with graph-structured stack to perform probabilistic parsing are discussed. Wright and Wrigley (1989) has proposed a probabilistic LR-table construction algorithm for non-left-recursive context-free grammars. To account for left recursions, a method for computing item probabilities using the generation of systems of linear equations is presented. The notion of deferred probabilities is proposed as a means for dealing with similar item sets with differing probability assignments.

1 Introduction

Probabilistic grammars provide a formalism which accounts for certain statistical aspects of the language, allows stochastic disambiguation of sentences, and helps in the efficiency of the syntactic analysis. Generalized LR parsing is a highly efficient parsing algorithm that has been adapted to handle arbitrary context-free grammars. To combine the advantages of both mechanisms, an algorithm for constructing a generalized probabilistic LR parser given a probabilistic context-free grammar is needed. In Wright and Wrigley (1989), a probabilistic LR-table construction method has been proposed for non-left-recursive context-free grammars. However, in practice, left-recursive context-free grammars are not uncommon, and it is often necessary to retain this left-recursive grammar structure. Thus, a method for handling left-recursions is needed in order to attain probabilistic LR-table construction for *general* context free grammars.

In this paper, we concentrate on incorporating probabilistic grammars with generalized LR parsing for efficiency. Stochastic information from probabilistic grammar can be used in making statistical decision during runtime to improve performance. In Section 3, we show how to adapt Tomita's(1985, 1987) generalized LR parser with graph-structured stack to perform probabilistic parsing and discuss related implementation issues. In Section 4, we describe the difficulty in computing item probabilities for left recursive context-free grammars. A solution is proposed in Section 5, which involves encoding item dependencies in terms of a system of linear equations. These equations can then be solved by Gaussian Elimination (Strang 1980) to give the item probabilities, from which the stochastic factors of the corresponding parse actions can be computed as described in Wright and Wrigley (1989).

We also introduce the notion of *deferred probability* in Section 6 in order to prevent creating excessive number of duplicate items which are similar except for their probability assignments.

*This research was supported in part by National Science Foundation under contract IRI-8858085.

2 Background

Probabilistic LR parsing is based on the notions of probabilistic context-free grammar and probabilistic LR parsing table, which are both augmented versions of their nonprobabilistic counterparts. In this section, we provide the definitions for the probabilistic versions.

2.1 Probabilistic CFG

A *probabilistic context-free grammar* (PCFG) (Suppes 1970, Wetherall 1980, Wright and Wrigley 1989) G, is a 4-tuple $\langle N, T, R, S \rangle$ where N is a set of non-terminal symbols including S the start symbol, T a set of terminal symbols, and R a set of probabilistic productions of the form $< A \rightarrow \alpha,\ p >$ where $A \in N$, $\alpha \in (N \cup T)^*$, and p the production probability. The probability p is the conditional probability $P(\alpha|A)$, which is the probability that the non-terminal A which appears during a derivation process is rewritten by the sequence α. Clearly if there are k A-productions with probabilities $p_1, \ldots, p_k$, then $\sum_{i=1}^{k} p_i = 1$, since the symbol A must be rewritten by the right hand side of some A-production. The production probabilities can be estimated from the corpus as outlined in Fu and Booth(1975) or Fujisaki(1984).

It is assumed that the steps of every derivation in the PCFG are mutually independent, meaning that the probability of applying a rewrite rule de-

Figure 1: GRA1: A Non-left Recursive PCFG

(1)	$S \rightarrow NP\ VP$	1
(2)	$NP \rightarrow n$	$\frac{1}{3}$
(3)	$NP \rightarrow det\ n$	$\frac{2}{3}$
(4)	$VP \rightarrow v\ NP$	1

Figure 2: GRA2: A Left-recursive PCFG

(1)	$S \rightarrow NP\ VP$	$\frac{3}{4}$
(2)	$S \rightarrow S\ PP$	$\frac{1}{4}$
(3)	$NP \rightarrow n$	$\frac{1}{2}$
(4)	$NP \rightarrow det\ n$	$\frac{2}{5}$
(5)	$NP \rightarrow NP\ PP$	$\frac{1}{10}$
(6)	$PP \rightarrow prep\ NP$	1
(7)	$VP \rightarrow v\ NP$	1

pends only upon the presence of a given nonterminal symbol (the premis) in a derivation and not upon how the premis was generated. Thus, the probability of a derivation is simply the product of the production probabilities of the productions in the derivation sequence.

Figures 1, 2 and 3 show three example PCFGs GRA1, GRA2 and GRA3 respectively. Incidentally, GRA1 is non-left recursive, GRA2 and GRA3 are both left-recursive, although GRA3 is "more" left-recursive than GRA2. GRA2 is said to have *simple recursion* since there is only a finite number of distinct left-recursive loops[1] in the grammar. GRA3, on the other hand, is said to have *massive left recursions* because of the intermingled left recursions, which result in infinite (possibly uncountable) number of distinct left-recursive loops in the grammar.

[1] A loop is a derivation cycle in which the first and last productions used in the derivation sequence are the same and occur nowhere else in the sequence.

Figure 3: GRA3:A Massively Left-recursive PCFG

(1)	$S \rightarrow S\ a_1$	$\frac{1}{7}$
(2)	$S \rightarrow B\ a_2$	$\frac{4}{7}$
(3)	$S \rightarrow C\ a_3$	$\frac{2}{7}$
(4)	$B \rightarrow S\ a_3$	$\frac{1}{3}$
(5)	$B \rightarrow B\ a_2$	$\frac{1}{6}$
(6)	$B \rightarrow C\ a_1$	$\frac{1}{2}$
(7)	$C \rightarrow S\ a_2$	$\frac{1}{5}$
(8)	$C \rightarrow B\ a_3$	$\frac{4}{15}$
(9)	$C \rightarrow C\ a_1$	$\frac{1}{15}$
(10)	$C \rightarrow a_3\ B$	$\frac{1}{3}$
(11)	$C \rightarrow a_3$	$\frac{2}{15}$

2.2 Probabilistic LR Parse Table

A probabilistic LR table is an augmented LR table of which the entries in the ACTION-table contains an additional field which is the probability of the action. We call this probability *stochastic factor* because it is the factor used in the computation (multiplication) of the *runtime stochastic product*. The parser keeps this stochastic product during runtime for each possible derivation, reflecting their respective likelihoods. This product can be computed during runtime by multiplication using the precomputed stochastic factors of the parsing actions (or by addition if the stochastic factors are expressed in logarithms). The parser can use this stochastic information to disambiguate or direct/prune its search probabilistically.

Figures 4, 5 and 6 show the respective probabilistic parsing tables for GRA1, GRA2 and GRA3, as constructed by the algorithm outlined in Section 5. Note that the stochastic factors of distinct actions associated with a state add up to 1 as expected, since each action's stochastic factor is simply the probability of the parser making that action during that point of parse. The format of the GOTO-table is unchanged as no stochastic factor is associated with GOTO actions.

3 Generalized Probabilistic LR Parsers for Arbitrary PCFGs

In this section, we describe how the efficient generalized LR parser with graph-structured stack in (Tomita 1985, 1987) can be adapted to parse probabilistically using the augmented parsing table. In particular, we discuss how to maintain consistent runtime stochastic products base on three key notions of the graph-structured stack: merging, local ambiguity packing and splitting. We assume that the state number and the respective runtime stochastic product are stored at each stack node.

3.1 Merging

Merging occurs when an element is being shifted onto two or more of the stack tops. Figure 7 illustrates a typical scenario in which a new state (State 3) is pushed onto stack tops States 1 and 2, of which original stochastic products are p_1 and p_2 respectively. These two nodes's stochastic products are modified to p_1q_1 and p_2q_2 correspondingly. If the stochastic factors of the actions has been represented as logarithms in the parse table, then their new "product" (or rather, logarithmic sums) would be $p_1 + q_1$ and $p_2 + q_2$ instead. For the stochastic product of Node 3, we can either use the sum of its parents' products (giving p_3 as $p_1q_1 + p_2q_2$) if we adopt *strict probabilistic approach*, or the maximum of the products (ie, $p_3 = \max(p_1q_1, p_2q_2)$) if we adopt the

Figure 4: Probabilistic Parsing Table for GRA1

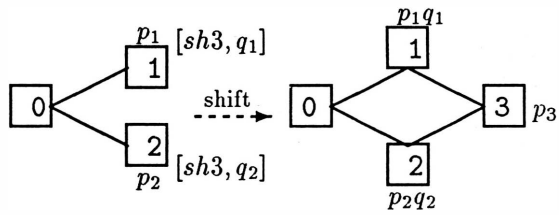
State	ACTION				GOTO		
	det	*n*	*v*	$	*NP*	*VP*	*S*
0	$\langle sh2, \frac{2}{3}\rangle$	$\langle sh1, \frac{1}{3}\rangle$			4		3
1			$\langle re2, 1\rangle$	$\langle re2, 1\rangle$			
2		$\langle sh5, 1\rangle$					
3				$\langle acc, 1\rangle$			
4			$\langle sh6, 1\rangle$			7	
5			$\langle re3, 1\rangle$	$\langle re3, 1\rangle$			
6	$\langle sh2, \frac{2}{3}\rangle$	$\langle sh1, \frac{1}{3}\rangle$			8		
7				$\langle re1, 1\rangle$			
8				$\langle re4, 1\rangle$			

Figure 5: Probabilistic Parsing Table for GRA2

State	ACTION					GOTO			
	det	*n*	*v*	*prep*	$	*NP*	*PP*	*VP*	*S*
0	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				3			4
1			$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$				
2		$\langle sh5, 1\rangle$							
3			$\langle sh7, \frac{9}{10}\rangle$	$\langle sh6, \frac{1}{10}\rangle$			8	9	
4				$\langle sh6, \frac{1}{4}\rangle$	$\langle acc, \frac{3}{4}\rangle$		10		
5			$\langle re4, 1\rangle$	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$				
6	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				11			
7	$\langle sh2, \frac{4}{9}\rangle$	$\langle sh1, \frac{5}{9}\rangle$				12			
8			$\langle re5, 1\rangle$	$\langle re5, 1\rangle$	$\langle re5, 1\rangle$				
9				$\langle re1, 1\rangle$	$\langle re1, 1\rangle$				
10				$\langle re2, 1\rangle$	$\langle re2, 1\rangle$				
11			$\langle re6, \frac{9}{10}\rangle$	$\langle re6, \frac{9}{10}\rangle$ $\langle sh6, \frac{1}{10}\rangle$	$\langle re6, \frac{9}{10}\rangle$		8		
12			$\langle re7, \frac{9}{10}\rangle$	$\langle re7, \frac{9}{10}\rangle$ $\langle sh6, \frac{1}{10}\rangle$	$\langle re7, \frac{9}{10}\rangle$		8		

maximum likelihood approach. Note that although the maximum likelihood approach is in some sense less "accurate" than the strict probabilistic approach, it is a reasonable approximate and has an added advantage when the stochastic factors are represented in logarithms, in which case the stochastic "products" of the parse stack can be maintained using only addition and subtraction operators(assuming, of course, that additions and subtractions are "cheaper" computationally than multiplications and divisions).

Figure 7: Merging



3.2 Local Ambiguity Packing

Local ambiguity packing occurs when two or more branches of the stack are reduced to the same non-terminal symbol. To be precise, this occurs when the parser attempts to create a GOTO state node (after a reduce action, that is) and realize that the parent already has a child node of the same state. In this case there is no need to create the GOTO node but to use that child node ("packing"). This is equivalent to the merging of shift nodes, and can be handled similarly: the runtime product of the child node is modified to the new "merged" product (either by summation or maximalization). This modification should be propagated accordingly to the successors of the packed child node, if any.

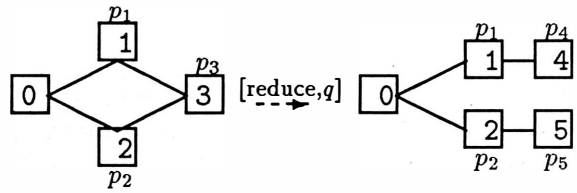
Figure 6: Probabilistic Parsing Table for GRA3

State	ACTION				GOTO		
	a_1	a_2	a_3	\$	S	B	C
0			$\langle sh1, 1\rangle$		2	3	4
1	$\langle re11, \frac{2}{7}\rangle$		$\langle re11, \frac{2}{7}\rangle$ $\langle sh1, \frac{5}{7}\rangle$		5	6	7
2	$\langle sh9, \frac{1}{7}\rangle$	$\langle sh8, \frac{33}{203}\rangle$	$\langle sh10, \frac{64}{203}\rangle$	$\langle acc, \frac{11}{29}\rangle$			
3		$\langle sh11, \frac{37}{48}\rangle$	$\langle sh12, \frac{11}{48}\rangle$				
4	$\langle sh13, \frac{107}{165}\rangle$		$\langle sh14, \frac{58}{165}\rangle$				
5	$\langle sh9, \frac{1}{7}\rangle$	$\langle sh8, \frac{66}{259}\rangle$	$\langle sh10, \frac{156}{259}\rangle$				
6	$\langle re10, \frac{77}{234}\rangle$	$\langle sh15, \frac{113}{234}\rangle$	$\langle re10, \frac{77}{234}\rangle$ $\langle sh12, \frac{22}{117}\rangle$				
7	$\langle sh16, \frac{128}{165}\rangle$		$\langle sh14, \frac{37}{165}\rangle$				
8	$\langle re7, 1\rangle$		$\langle re7, 1\rangle$				
9	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$	$\langle re1, 1\rangle$			
10	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$	$\langle re4, 1\rangle$				
11	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$ $\langle re5, \frac{8}{37}\rangle$	$\langle re2, \frac{29}{37}\rangle$			
12	$\langle re8, 1\rangle$		$\langle re8, 1\rangle$				
13	$\langle re6, \frac{96}{107}\rangle$ $\langle re9, \frac{11}{107}\rangle$	$\langle re6, \frac{96}{107}\rangle$	$\langle re6, \frac{96}{107}\rangle$ $\langle re9, \frac{11}{107}\rangle$				
14	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$	$\langle re3, 1\rangle$			
15	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$ $\langle re5, \frac{39}{113}\rangle$	$\langle re2, \frac{74}{113}\rangle$			
16	$\langle re6, \frac{117}{128}\rangle$ $\langle re9, \frac{11}{128}\rangle$	$\langle re6, \frac{117}{128}\rangle$	$\langle re6, \frac{117}{128}\rangle$ $\langle re9, \frac{11}{128}\rangle$				

3.3 Splitting

Splitting occurs when there is an action conflict. This can be handled straightforwardly by creating corresponding new nodes for the new resulting states with the respective runtime products (such as the product of the parent's stochastic product with the action's stochastic factor). Splitting can also occur when reducing (popping) a merged node. In this case, the parser needs to recover the original runtime product of the merged components, which can be obtained with some mathematical manipulation from the runtime products recorded in the merged node's parents. Figure 8 illustrates a simple situation in which a merged node is split into two. In the figure, a reduce action (of which the corresponding production is of unit length) is applied at Node 3, and the GOTO's for Nodes 1 and 2 are states 4 and 5 respectively. In the case that strict probabilistic approach is used in merging (see above), we get $p_4 = \frac{p_1}{p_1+p_2} p_3 q$ and $p_5 = \frac{p_2}{p_1+p_2} p_3 q$. If the maximum likelihood approach is used, then $p_4 = \frac{p_1}{\max(p_1,p_2)} p_3 q$ and $p_5 = \frac{p_2}{\max(p_1,p_2)} p_3 q$. Furthermore, if the stochastic factors have been expressed in logarithms, then $p_4 = p_3 - \max(p_1, p_2) + p_1 + q$ and $p_5 = p_3 - \max(p_1, p_2) + p_2 + q$ (notice that only addition and subtraction are needed, as promised).

Figure 8: Splitting



In general, there may be more than one splitting corresponding to a reduce action (ie, we may have to pop more than one merged nodes). For every split node, we must recover the runtime products of its parents to obtain the appropriate stochastic products for the resulting new branches. This can be tricky and is one of the reasons why a tree-structured stack (described below) instead of graphs might perform better in some cases.

3.4 Using Stochastic Product to Guide Search

The main point of maintaining the runtime stochastic products is to use it as a good indicator

function to guide search. In practical situation, the grammar can be highly ambiguous, resulting in many branches of ambiguity in the parse stack. As discussed before, the runtime stochastic product reflects the likelihood of that branch to complete successfully.

In Tomita's generalized LR parser, processes are synchronized by performing all the reduce actions before the shift actions. In this way, the processes are made to scan the input at the same rate, which in turn allows the unification of processes in the same state. Thus, the runtime stochastic products can be a good enough indicator of how promising each branch (ie. partial derivation) is, since we are comparing among partial derivations of same input length. We can perform beam search by pruning away branches which are less promising.

If instead of the breadth-first style beam search approach described above we employ a best-first (or depth-first) strategy, then not all of the branches will correspond to the same input length. Since the measure of runtime stochastic product is biased towards shorter sentences, a good heuristic would have to take into account of the number of input symbols consumed. Even so, handling best-first search can be tricky with Tomita's graph-structured stack without the process-input synchronization, especially with the merging and packing of nodes. Presumably, we can have additional data structure to serve as lookup table of the nodes currently in the graph stack: for instance, an n by m matrix (where n is the number of states in the parse table and m the input length) indexed by the state number and the input position storing pointers to current stack nodes. With this lookup table, the parser can check if there is any stack node it can use before creating a new one. However, in the worst case, the nodes that could have been merged or packed might have already been popped of the stack before it can be re-used. In this case, the parser degenerates into one with tree-structured stack (ie, only splitting, but no merging and packing) and the laborious book-keeping of the stochastic products due to the graph structure of the parse stack seems wasted. It might be more productive then to employ a tree-structured stack instead of a graph-structured stack, since the book-keeping of runtime stochastic products for trees is much simpler: as each tree branch represents exactly one possible parse, we can associate the respective runtime stochastic products to the leaf nodes (instead of every node) in the parse stack, and updating would involve only multiplying (or adding, in the logarithmic case) with the stochastic factors of the corresponding parse actions to obtain the new stochastic products. The major drawback of the tree-stack version is that it is merely a slightly compacted form of stack list (Tomita 1987) — which means that the tree can grow unmanageably large in a short period, unless suitable pruning is done. Hopefully, the runtime stochastic product will serve as good heuristic for pruning the branches; but whether it is the case that the simplicity of the tree implementation overrides that of the representational efficiency of the graph version remains to be studied.

4 Problem with Left Recursion

The approach to probabilistic LR table construction for non-left recursive PCFG , as proposed by Wright and Wrigley(1989), is to augment the standard SLR table construction algorithm presented in Aho and Ullman(1977) to generate a probabilistic version. The notion of a probabilistic item $\langle A \rightarrow \alpha \cdot \beta,\ p \rangle$ is introduced, with $\langle A \rightarrow \alpha \cdot \beta \rangle$ being an ordinary LR(0) item, and p the item probability, which is interpreted as the posterior probability of the item in the state. The major extension is the computation of these item probabilities from which the stochastic factors of the parse actions can be determined. Wright and Wrigley(1989) have shown a direct method for computing the item probabilities for non-left recursive grammars. The probabilistic parsing table in Figure 4 for the non-left recursive grammar GRA1 is thus constructed.

Since there is an algorithm for removing left recursions from a context-free grammar (Aho and Ullman 1977), it is conceivable that the algorithm can be modified to convert a left-recursive PCFG to one that is non left-recursive. Given a left-recursive PCFG, we can apply this algorithm, and then use Wright and Wrigley(1989)'s table construction method on the resulting non left-recursive grammar to create the parsing table. Unfortunately, the left-recursion elimination algorithm destructs the original grammar structure. In practice, especially in natural language processing, it is often necessary to preserve the original grammar structure. Hence a method for constructing a parse table without grammar conversion is needed.

For grammars with left recursion, the computation of item probabilities becomes nontrivial. First of all, item probability ceases to be a "probability", as an item which is involved in left recursion is effectively a coalescence of an infinite number of similar items along the cyclic paths, so its associated stochastic value is the sum of posteriori probabilities of these packed items. For instance, if starting from item $\langle A \rightarrow \alpha \cdot B\beta,\ p \rangle$ we derive the item $\langle C \rightarrow \cdot B\gamma,\ p \times p_B \rangle$, then by left recursion we must also have the items $\langle C \rightarrow \cdot B\gamma,\ p \times p_B^i \rangle$ for $i = 1, \ldots \infty$. The probabilistic item $\langle C \rightarrow \cdot B\gamma,\ q \rangle$, being a coalescence of these items, would have item probability $q = \sum_{i=1}^{\infty} p \times p_B^i = \frac{p}{1-p_B}$,

and there is no guarantee that $q \leq 1$. This is understandable since $\langle C \rightarrow \cdot B\gamma, q \rangle$ is a coalescence of items which are not necessarily mutually exclusive. However, we need not be alarmed as the stochastic values of the underlying items are still legitimate probabilities.

Owing to this coalescence of infinite items into one single item in left recursive grammars, the computation of the stochastic values of items involves finding infinite sums of the items' stochastic values. For grammars with simple left recursion (that is, there are only finitely many left recursion loops) such as GRA2, we can still figure out the sum by enumeration, since there is only a finite number of the infinite sums corresponding to the left recursion loops. With massive left recursive grammars like GRA3 in which there is an infinite number of (intermingled) left recursion loops, the enumeration method fails. We shall illustrate this effect in the following sections.

4.1 Simple Left Recursion

For grammars with simple left recursion, it is possible to derive the stochastic values by simple cycle detection. For instance, consider the following set of LR(0) items for GRA2 in Figure 9.

Figure 9: An Example State for GRA2

I_0:	$[VP \rightarrow v \cdot NP,$	$S_0]$
I_1:	$[NP \rightarrow \cdot n,$	$S_1]$
I_2:	$[NP \rightarrow \cdot det\ n,$	$S_2]$
I_3:	$[NP \rightarrow \cdot NP\ PP,$	$S_3]$

Suppose the kernel set contains only I_0, with $S_0 = \frac{3}{7}$. Let $\mathcal{D}$ be a partial derivation before seeing the input symbol v. At this point, the possible derivations which will lead to item I_1 are:

$\mathcal{D} \overset{S_0}{\Longrightarrow} VP \rightarrow v \cdot NP \overset{\frac{1}{2}}{\Longrightarrow} NP \rightarrow \cdot n$

$\mathcal{D} \overset{S_0}{\Longrightarrow} VP \rightarrow v \cdot NP \overset{\frac{1}{10}}{\Longrightarrow} NP \rightarrow \cdot NP\ VP \overset{\frac{1}{2}}{\Longrightarrow} NP \rightarrow \cdot n$

$\vdots$

$\mathcal{D} \overset{S_0}{\Longrightarrow} VP \rightarrow v \cdot NP \overset{\frac{1}{10}}{\Longrightarrow} NP \rightarrow \cdot NP\ VP \overset{\frac{1}{10}}{\Longrightarrow} \ldots \overset{\frac{1}{2}}{\Longrightarrow} NP \rightarrow \cdot n$

$\vdots$

The sum of the posterior probabilities of the above possible partial derivations are:

$S_1 = (S_0 \times \frac{1}{2}) + (S_0 \times \frac{1}{10} \times \frac{1}{2}) + (S_0 \times \frac{1}{10}^2 \times \frac{1}{2}) + \ldots$
$= \frac{3}{7} \times \sum_{n=0}^{\infty} \frac{1}{10}^n \times \frac{1}{2} = \frac{5}{21}$

Similarly, $S_2 = \frac{3}{7} \times \sum_{n=0}^{\infty} \frac{1}{10}^n \times \frac{2}{5} = \frac{4}{21}$, and $S_3 = \frac{3}{7} \times \sum_{n=1}^{\infty} \frac{1}{10}^n = \frac{1}{21}$.

4.2 Massive Left Recursion

For grammars with intermingled left recursions such as GRA3, computation of the stochastic values of the items becomes a convoluted task. Consider the start state for GRA3, which is depicted in Figure 10.

Figure 10: Start State of GRA3

I_0:	$[S' \rightarrow \cdot S,$	$1]$
I_1:	$[S \rightarrow \cdot Sa_1,$	$S_1]$
I_2:	$[S \rightarrow \cdot Ba_2,$	$S_2]$
I_3:	$[S \rightarrow \cdot Ca_3,$	$S_3]$
I_4:	$[B \rightarrow \cdot Sa_3,$	$S_4]$
I_5:	$[B \rightarrow \cdot Ba_2,$	$S_5]$
I_6:	$[B \rightarrow \cdot Ca_1,$	$S_6]$
I_7:	$[C \rightarrow \cdot Sa_2,$	$S_7]$
I_8:	$[C \rightarrow \cdot Ba_3,$	$S_8]$
I_9:	$[C \rightarrow \cdot Ca_1,$	$S_9]$
I_{10}:	$[C \rightarrow \cdot a_3 B,$	$S_{10}]$
I_{11}:	$[C \rightarrow \cdot a_3,$	$S_{11}]$

Consider the item I_1. In an attempt to write down a closed expression for the stochastic value S_1, we discover in despair that there is an infinite number of loops to detect, as S is immediately reachable by all non-terminals, and so are the other nonterminals themselves. This intermingling of the loops renders it impossible to write down closed expressions for S_1 through S_{11}.

5 Probabilistic Parse Table Construction for Left Recursive Grammars

In this section, we describe a way of computing item probabilities by encoding the item dependencies in terms of systems of linear equations and solving them by Gaussian Elimination (Strang 1980). This method handles arbitrary context-free grammar including those with left recursions. We incorporate this method with Wright and Wrigley's(1989) algorithm for computing stochastic factors for the parse actions to obtain a table construction algorithm which handles general PCFG. A formal description of the complete table construction algorithm is in the Appendix.

In the following discussion of the algorithm, lower case greek characters such as α and β will denote strings in $(N \cup T)^*$ and upper case alphabets like A and B denote symbols in N unless mentioned otherwise.

5.1 Stochastic Values of Kernel Items

For completeness, we mention briefly here how the stochastic values of items in the kernel set can be computed as proposed by Wright and Wrigley(1989):

The stochastic value of the kernel item $[S' \rightarrow \cdot S]$ in the start state is 1. Let State $m-1$ be a prior

state of the non-start State m. We want to compute the stochastic values of the kernel items of State m. Suppose in State $m-1$ there are k items which are expecting the grammar symbol X, their stochastic values being $S_1, S_2, \ldots, S_k$ respectively. Let $[A_i \rightarrow \alpha_i \cdot X\beta_i, S_i]$ be these item, $i = 1, \ldots, k$. Then the posterior probability of the kernel item $[A_i \rightarrow \alpha_i X \cdot \beta_i]$ of State m given those k items in State i and grammar symbol X as the next symbol seen on the parse stack is $\frac{S_i}{S_X}$, where $S_X = \sum_{i=1}^{k} S_i$.

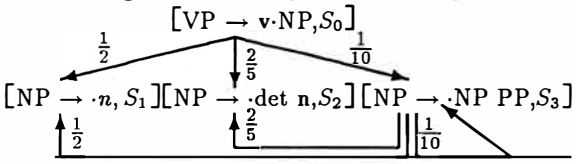
5.2 Dependency Graph

The inter-dependency of items within a state can be represented most straightforwardly by a dependency forest. If we label each arc by the probability of the rule represented by that item the arc is pointing at, then the posterior probability of an item in a dependency forest is simply the total product of the root item's stochastic value and the arc costs along the path from the root to the item.

This dependency forest can be compacted into a dependency graph in which no item occurs in more than one node. That is, each graph node represents a stochastic item which is a coalesce of all the nodes in the dependency forest representing that particular item. The stochastic value of such an item is thus the sum of the posterior probabilities of the underlying items.

Figure 11 depicts the graphical relations of the items in the example state of GRA2 in Figure 9. We shall not attempt to depict the massively cyclic dependency graph of the start state for GRA3 (Figure 10) here.

Figure 11: A Dependency Graph



5.3 Generating Linear Equations

Rather than attempting to write down a closed expression for the stochastic value of each item, we resort to creating a system of linear equations in terms of the stochastic values which encapsulate the possibly cyclic dependency structure of the items in the set.

Consider a state Ψ with k items, m of which are kernel items. That is, Ψ is the set of items $\{I_j \mid 1 \leq j \leq k\}$ such that I_j is a kernel item if $1 \leq j \leq m$. Again, let S_j be a variable representing the stochastic value of item I_j. The values of $S_1, \ldots, S_m$ are known since they can be computed as outlined in Section 5.1.

Consider a non-kernel item I_j, $m < j \leq k$. Let $\{I_{j_1}, \ldots, I_{j_{n'}}\}$ be the set of items in Ψ from which there is an arc into I_j in the dependency graph for Ψ. Also, let P_{j_i} denote the arc cost of the arc from item I_{j_i} to I_j. Then, the equation for the stochastic value of I_j, namely S_j, would be:

$$S_j = \sum_{i=1}^{n'} P_{j_i} \times S_{j_i} \qquad (1)$$

Note that Equation (1) is a linear equation of at most $(k-m)$ unknowns, namely $S_{m+1}, \ldots, S_k$. This means that from 1 we have a system of $(k-m)$ linear equations with $(k-m)$ unknowns. This can be solved using standard algorithms like simple Gaussian Elimination (Strang 1980).

The task of generating the equations can be further simplified by the following observations:

1. The cost of any incoming arc of a non-kernel item $I_i = [A_i \rightarrow \cdot\alpha_i, S_i]$ is the production probability of the production $\langle A_i \rightarrow \alpha_i, P_r \rangle$. In other words, $P_{j_i} = P_r$ for $i = 1 \ldots n'$. Equation (1) can then be simplified to $S_j = P_r \times \sum_{i=1}^{n'} S_{j_i}$.
2. Within a state, the non-kernel items representing any X-production have the same set of items with arcs into them. Therefore, these non-kernel items have the same value for $\sum_{x=1}^{n'} S_{j_x}$ (which is similar to the S_X in Section 5.1).

Thus, Equation (1) can be further simplified as $S_j = P_r \times S_{A_j}$ where $S_{A_j} = \sum_{x=1}^{n'} S_{j_x}$. With that, the system of linear equations for each state can be generated efficiently without having to construct explicitly the item dependency graph.

5.3.1 Examples

The system of linear equations for the state depicted in Figures 9 and 11 for grammar GRA2 is as follows:

$$S_0 = \tfrac{3}{7} \quad (Given) \qquad S_2 = \tfrac{2}{5}(S_0 + S_3)$$
$$S_1 = \tfrac{1}{2}(S_0 + S_3) \qquad S_3 = \tfrac{1}{10}(S_0 + S_3)$$

On solving the equations, we have $S_1 = \frac{5}{21}$, $S_2 = \frac{4}{21}$ and $S_3 = \frac{1}{21}$, which is the same solution as the one obtained by enumeration (Section 4.1).

Similarly, the following system of linear equations is obtained for the start state of massively left recursive grammar GRA3:

$$S_0 = 1 \qquad S_6 = \tfrac{1}{2}(S_2 + S_5 + S_8)$$
$$S_1 = \tfrac{1}{7}(S_0 + S_1 + S_4 + S_7) \qquad S_7 = \tfrac{1}{5}(S_3 + S_6 + S_9)$$
$$S_2 = \tfrac{4}{7}(S_0 + S_1 + S_4 + S_7) \qquad S_8 = \tfrac{4}{15}(S_3 + S_6 + S_9)$$
$$S_3 = \tfrac{2}{7}(S_0 + S_1 + S_4 + S_7) \qquad S_9 = \tfrac{1}{15}(S_3 + S_6 + S_9)$$
$$S_4 = \tfrac{1}{3}(S_2 + S_5 + S_8) \qquad S_{10} = \tfrac{1}{3}(S_3 + S_6 + S_9)$$
$$S_5 = \tfrac{1}{6}(S_2 + S_5 + S_8) \qquad S_{11} = \tfrac{2}{15}(S_3 + S_6 + S_9)$$

On solving the equations, we have the solutions 1, $\frac{29}{77}$, $\frac{116}{77}$, $\frac{58}{77}$, $\frac{64}{77}$, $\frac{32}{77}$, $\frac{96}{77}$, $\frac{3}{7}$, $\frac{4}{7}$, $\frac{1}{7}$, $\frac{5}{7}$ and $\frac{2}{7}$ for the stochastic variables S_0 through S_{11} respectively.

5.4 Solving Linear Equations with Gaussian Elimination

The systems of linear equations generated during table construction can be solved using the popular method *Gaussian Elimination* which can be found in many numerical analysis or linear algebra textbooks (for example, Strang 1980) or linear programming books (such as Vašek Chvátal, 1983). The basic idea is to eliminate the variables one by one by repeated substitutions. For instance, if we have the following set of equations:

(1) $S_1 = a_{11}S_1 + a_{12}S_2 + \ldots + a_{1n}S_n$

$\vdots$

(n) $S_n = a_{n1}S_1 + a_{n2}S_2 + \ldots + a_{nn}S_n$

We can eliminate S_1 and remove equation (1) from the system by substituting, for all occurrences of S_1 in equations (2) through (n), the right hand side of equation (1). We repeatedly remove variables S_1 through S_{n-1} in the same way, until we are left with only one equation with one variable S_n. Having thus obtained the value for S_n, we perform back substitutions until solutions for S_1 through S_n are obtained.

Complexity-wise, Gaussian elimination is a cubic algorithm(Vašek Chvátal, 1983) in terms of the number of variables (ie, the number of items in the closure set). The generation of linear equations per state is also polynomial since we only need to find the stochastic sum expressions — the S_{A_i}'s, for the nonterminals (Point 2 of Section 5.3). These expressions can be obtained by partitioning the items in the state set according to their left hand sides. There are $O(mn)$ possible LR(0) items (hence the size of each state is $O(mn)$) and $O(2^{mn})$ possible sets where n is the number of productions and m the length of the longest right hand side. Hence, asymptotically, the computation of the stochastic values would not affect the complexity of the algorithm, since it has only added an extra polynomial amount of work for each of the exponentially many possible sets.

Of course, we could have used other methods for solving these linear equations, for example, by finding the inverse of the matrix representing the equations(Vašek Chvátal, 1983). It is also plausible that particular characteristics of the equations generated by the construction algorithm can be exploited to derive the equations' solution more efficiently. We shall not discuss further here.

5.5 Stochastic Factors

Since the stochastic values of the terminal items in a parse state are basically posterior probabilities of that item given the root (kernel) item, the computation of the stochastic factors for the parsing actions, which is as presented in Wright and Wrigley(1989), is fairly straightforward. For *shift*-action, say from State i to State $i+1$ on seeing the input symbol x, the corresponding stochastic factor for this action would be S_x, the sum of the stochastic values of all the leaf items in State i which are expecting the symbol x. For *reduce*-action, the stochastic factor is simply the stochastic value S_i of the item representing the reduction, namely $[A_i \rightarrow \alpha_i \cdot, S_i]$ if the reduction is via production $A_i \rightarrow \alpha_i$. For *accept*-action, the stochastic factor is the stochastic value S_n of the item $[S' \rightarrow S \cdot, S_n]$, since acceptance can be treated as a final reduction of the augmented production $S' \rightarrow S$, where S' is the system-introduced start symbol for the grammar.

6 Deferred Probabilities

The introduction of probability created a new criterion for equality between two sets of items: not only must they contain the same items, they must have the same item probability assignment. It is thus possible that we have many (possibly infinite) sets of similar items of differing probability assignments. This is especially so when there are loops amongst the sets of items (ie, the *states*) in the automaton created by the table construction algorithm — there is no guarantee that the differing probability assignments of the recurring states would converge. Even if they do converge eventually, it is still undesirable to have a huge parsing table of which many states have exactly the same underlying item set but differing probabilities.

To remedy this undesirable situation, we introduce a mechanism called ***deferred probability*** which will guarantee that the item sets converge without duplicating too many of the states. Thus far, we have been precomputing item's stochastic values in an *eager* fashion — propagating the probabilities as early as possible. Deferred probability provides a means to defer propagating certain problematic probability assignments (problematic in the sense that it causes many similar states with differing probability assignments) until appropriate. In the extreme case, probabilities are deferred until *reduction* time, ie, the stochastic factors of REDUCE actions are the respective rule probabilities and all other parse actions have unit stochastic factors. A reasonable postponement, however, would be to defer propagating the probabilities of the kernel items (kernel probabilities) until the following state. By forcing the differing item sets to have some fixed predefined probability assignment (while deferring the propagation of the "real" probabilities until appropriate times), we can prevent excessive duplication of

similar states with same items but different probabilities.

To allow for deferred probabilities, we extend the original notion of probabilistic item to contain an additional field q which is the deferred probability for that item. That is, a probabilistic item would have the form $\langle A \rightarrow \alpha \cdot \beta,\ p,\ q\rangle$. The default value of q is 1, meaning that no probability has been deferred. If in the process of constructing the closure states the table-construction program discovers that it is re-creating many states with the same underlying items but with differing probabilities or when it detects a non-converging loop, it might decide to replace that state with one in which the original kernel probabilities are deferred. That is, if the item $\langle A \rightarrow \alpha \cdot \beta,\ p,\ q\rangle$ is a kernel item, and $\beta \neq \epsilon$, we replace it with a deferred item $\langle A \rightarrow \alpha \cdot \beta,\ p',\ \frac{pq}{p'}\rangle$ and proceed to compute the closure of the kernel set as before (ie, ignoring the deferred probabilities). In essence we have reassigned a kernel probability of p' to the kernel items *temporarily* instead of its original probability. It is important that this choice of assignment of p' be fixed with respect to that state. For instance, one assignment would be to impose a uniform probability distribution onto the deferred kernel items, that is, let p' be the probability $\frac{1}{Number\ of\ kernel\ items}$. Another choice is to assign unit probability to each of the kernel items, which allows us to simulate the effect of treating each of the kernel items as if it forms a separate state.

Although in theory it is possible to defer the kernel probabilities until reduction time, in practice it is sufficient to defer it for only one state transition. That is, we recover the deferred probabilities in the next state. We can do this by enabling the propagation of the deferred probabilities in the next state, simply by multiplying back the deferred probabilities q into the kernel probabilities of the next state. In other words, as in Section 5.1, if $[A_i \rightarrow \alpha_i \cdot X\beta_i, S_i, q]$ is in State $m-1$, then the corresponding kernel item in State m would be $[A_i \rightarrow \alpha_i X \cdot \beta_i, \frac{S_i q}{S_X}, 1]$.

7 Concluding Remarks

In this paper, we have presented a method for dealing with left recursions in constructing probabilistic LR parsing tables for left recursive PCFGs. We have described runtime probabilistic LR parsers which use probabilistic parsing table. The table construction method, as outlined in this paper and more formally in the appendix, has been implemented in Common Lisp. The two versions of runtime parsers described in this paper have also been implemented in Common Lisp, and incorporated with various search strategies such as beam-search and best-first search (only for the tree-stack version) for comparison. The programs run successfully on various small toy grammars, including the ones listed in this paper. In future, we hope to experiment with larger grammars such as the one in Fujisaki(1984).

Appendix A. Table Construction Algorithm

A full algorithm for probabilistic LR parsing table construction for general probabilistic context-free grammar is presented here. The deferred probability mechanism as described in Section 6 is employed, the chosen reassignment of kernel probability being the unit probability.

A.1 Auxiliary Functions

A.1.1 CLOSURE

CLOSURE takes a set of ordinary nonprobabilistic LR(0) items and returns the set of LR(0) items which is the closure of the input items. A standard algorithm for CLOSURE can be found in Aho and Ullman(1977).

A.1.2 PROB-CLOSURE

Input: A set of k probabilistic items for some $k \geq 1$: $\{[A_i \rightarrow \alpha_i \cdot \beta_i, p_i, q_i] \mid 1 \leq i \leq k\}$.

Output: A set of probabilistic items which is the closure of the input probabilistic items. Each probabilistic item in the output set carries a stochastic value which is the sum of the posterior probabilities of that item given the input items.

Method:

Step 1: Let
$\mathcal{C} :=$ CLOSURE($\{[A_i \rightarrow \alpha_i \cdot \beta_i] \mid 1 \leq i \leq k\}$);

Step 2: Suppose k' is the size of $\mathcal{C}$. Let I_i be the i-th item $[A_i \rightarrow \alpha_i.\beta_i]$ in $\mathcal{C}$, $1 \leq i \leq k'$. Also, for each item I_i, let S_i be a variable denoting its stochastic value.

1. For $1 \leq i \leq k$, $S_i := p_i$;
2. Let $\mathcal{E}_B$ be the set of items in $\mathcal{C}$ that are expecting B as the next symbol on the stack. That is, $\mathcal{E}_B$ is the set

$$\{I_j \mid I_j \in \mathcal{C},\ I_j = [A_j \rightarrow \alpha_j \cdot B\beta_j]\}$$

Let $S_B \stackrel{\text{def}}{=} \sum_{I_j \in \mathcal{E}_B} S_j$, where $B \in N$. For $k < i \leq k'$ such that $I_i = [A_i \rightarrow \cdot\beta i]$, set $S_i := P_r \times S_{A_i}$, where P_r is the probability of the production $A_i \rightarrow \beta_i$.

Step 3: Solve the system of linear equations generated by **Step 2**, using any standard algorithm such as simple Gaussian Elimination (Strang 1980).

Step 4: Return $\{[A_i \rightarrow \alpha \cdot \beta, S_i, q_i] \mid 1 \leq i \leq k'\}$, where $q_i = 1$ for $k \leq i \leq k'$.

A.1.3 GOTO

Another useful function in table construction is GOTO$(\{I_1 \ldots I_n\}, X)$, where the first argument $\{I_1 \ldots I_n\}$ is a set of n probabilistic items and the second argument X a grammar symbol in $(N \cup T)$.

Suppose the probabilistic items in $\{I_1 \ldots I_n\}$ are such that those with symbol X after the dot are $[A_i \rightarrow \alpha_i \cdot X\beta_i, S_i, q_i]$, $1 \leq i \leq k$ for some $1 \leq k \leq n$. Let S_X be $\sum_{i=1}^{k} S_i$ and set GOTO$(\{I_i\}, X)$ to be PROB-CLOSURE$(\{[A_i \rightarrow \alpha_i X \cdot \beta_i, \frac{S_i q_i}{S_X}, 1] \mid 1 \leq i \leq k\})$.

When $k = 0$, GOTO$(\{I_i\}, X)$ is undefined.

A.1.4 Sets-of-Items Construction

Let $\mathcal{U}$ be the canonical collection of sets of probabilistic items for the grammar G'. $\mathcal{U}$ can be constructed as described below.

Initially $\mathcal{U}$:= PROB-CLOSURE$(\{[S' \rightarrow \cdot S, 1]\})$. Repeat the process of applying the GOTO function (as defined in Step A.1.3) with the existing sets in $\mathcal{U}$ and symbols in $(N \cup T)$ to generate new sets to be added to $\mathcal{U}$. If it is detected that an excessive number of states with similar underlying item sets but differing probabilities are created, use a state that is created by deferring the probabilities of the kernel items. That is, suppose the original kernel set is $\{[A_i \rightarrow \alpha_i \cdot \beta_i, p_i, q_i] \mid 1 \leq i \leq k\}$, use instead $\{[A_i \rightarrow \alpha_i \cdot \beta_i, 1, p_i q_i] \mid 1 \leq i \leq k \text{ and } \beta_i \neq \epsilon\}$.

The process stops when no new set can be generated.

Note that equality between two sets of probabilistic items here requires that they contain the same items with equal corresponding stochastic values, as well as deferred probabilities.

A.2 LR Table Construction

The algorithm is very similar to standard LR table construction (Aho and Ullman 1977) except for the additional step to compute the stochastic factor for each action (*shift*, *reduce*, or *accept*).

Given a grammar $G = \langle N, T, R, S \rangle$, we define a corresponding grammar G' with a system-generated start symbol S':

$$\langle N \cup \{S'\}, T, R \cup \{< S' \rightarrow S, 1 >\}, S' \rangle.$$

Input: $\mathcal{U}$, the canonical collection of sets of probabilistic items for grammar G'.

Output: If possible, a probabilistic LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

Method: Let $\mathcal{U} = \{\Psi_0, \Psi_1, \ldots, \Psi_n\}$, where Ψ_0 is that initial set in Sets-of-Items Construction. The states of the parser are then $0, 1, \ldots, n$, with state i being constructed from Ψ_i. The parsing actions for state i are determined as follows:

1. If $[A \rightarrow \alpha \cdot a\beta, q_a]$ is in Ψ_i, $a \in T$, and GOTO$(\Psi_i, a) = \Psi_j$, set ACTION$[i, a]$ to $\langle$ "*shift* j", $p_a\rangle$ where p_a is the sum of q_a's – that is the stochastic values of items in Ψ_i with symbol a after the dot.
2. If $[A \rightarrow \alpha \cdot, p]$ is in Ψ_i, set ACTION$[i, a]$ to $\langle$ "*reduce* $A \rightarrow \alpha$", $p\rangle$ for every $a \in$ FOLLOW(A).
3. If $[S' \rightarrow S \cdot, p]$ is in Ψ_i, set ACTION$[i, \$]$ ($\$$ is an end-of-input marker) to $\langle$ "*accept*", $p\rangle$.

The goto transitions for state i are constructed in the usual way:

4. If GOTO$(I_i, A) = I_j$, set GOTO$[i, A] = j$

All entries not defined by rules (1) through (4) are made "*error*".

The FOLLOW table can be constructed from G by a standard algorithm in Aho and Ullman(1977).

References

Aho, A.V. and Ullman, J.D. 1977. *Principles of Compiler Design.* Addison Wesley.

Fu, K. S. , and Booth, T. L. , 1975. Grammatical Inference: Introduction and Survey — Part II. *IEEE Trans on Sys., Man and Cyber.* SMC-5:409-423.

Fujisaki, T. 1984. An Approach to Stochastic Parsing. *Proceedings of COLING84.*

Strang, G. 1980. *Linear Algebra and Its Applications*, 2nd Ed. Academic Press, New York, NY.

Suppes, P. 1970. Probabilistic Grammars for Natural Languages. *Synthese* 22:95-116.

Tomita, M. 1985. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, MA.

Tomita, M. January-June, 1987. An Efficient Augmented Context-Free Parsing Algorithm. *Computational Linguistics* 13(1-2):31-46.

Vašek Chvátal, 1983. *Linear Programming*, Chapter 6.

Wetherall, C. S. 1980. Probabilistic Languages: A Review and Some Open Questions. *Computing Surveys* 12:361-379.

Wright, J.H. and Wrigley, E.N. 1989. Probabilistic LR Parsing for Speech Recognition. *International Parsing Workshop '89*, Carnegie Mellon University, Pittsburgh PA.

February 15, 1991

Session A

Quasi-Destructive Graph Unification

Hideto Tomabechi

ATR Interpreting Telephony Research Laboratories*
Seika-cho, Sorakugun, Kyoto 619-02 JAPAN

Carnegie Mellon University
109 EDSH, Pittsburgh, PA 15213-3890, USA
tomabech@a.nl.cs.cmu.edu

ABSTRACT

Graph unification is the most expensive part of unification-based grammar parsing. It often takes over 90% of the total parsing time of a sentence. We focus on two speed-up elements in the design of unification algorithms: 1) elimination of excessive copying by only copying successful unifications, 2) Finding unification failures as soon as possible. We have developed a scheme to attain these two criteria without expensive overhead through temporarily modifying graphs during unification to eliminate copying during unification. The temporary modification is invalidated in constant time and therefore, unification can continue looking for a failure without the overhead associated with copying. After a successful unification because the nodes are temporarily prepared for copying, a fast copying can be performed without overhead for handling reentrancy, loops and variables. We found that parsing relatively long sentences (requiring about 500 unifications during a parse) using our algorithm is 100 to 200 percent faster than parsing the same sentences using Wroblewski's algorithm.

1. Motivation

Graph unification is the most expensive part of unification-based grammar parsing systems. For example, in the three types of parsing systems currently used at ATR[1], all of which use graph unification algorithms based on [Wroblewski, 1987], unification operations consume 85 to 90 percent of the total cpu time devoted to a parse. The number of unification operations per sentence tends to grow as the grammar gets larger and more complicated. An unavoidable paradox is that when the natural language system gets larger and the coverage of linguistic phenomena increases the writers of natural language grammars tend to rely more on deeper and more complex path equations (loops and frequent reentrancy) to lessen the complexity of writing the grammar. As a result, we have seen that the number of unification operations increases rapidly as the coverage of the grammar grows in contrast to the parsing algorithm itself which does not seem to grow so quickly. Thus, it makes sense to speed up the unification operations to improve the total speed performance of the natural language parsing system.

Our original unification algorithm was based on [Wroblewski, 1987] which was chosen in 1988 as the then fastest algorithm available for our application (HPSG based unification grammar, three types of parsers (Earley, Tomita-LR, and active chart), unification with variables and loops[2] combined with Kasper's ([Kasper, 1987]) scheme for handling disjunctions). In designing the graph unification algorithm, we have made the following observation which influenced the basic design of the new algorithm described in this paper:

Unification does not always succeed.

As we will see from the data presented in a later section, when our parsing system operates with a relatively small grammar, about 60 percent of unifications attempted during a successful parse result in failure. If a unification fails, any computation performed and memory consumed during the unification is wasted. As the grammar size increases, the number of unification failures for each successful parse increases[3]. Without completely rewriting the grammar and the parser, it seems difficult to shift any significant amount of the computational burden to the parser in order to reduce the number of unification failures[4].

Another problem that we would like to address in our design, which seems to be well documented in the existing literature is that:

Copying is an expensive operation.

The copying of a node is a heavy burden to the parsing system. [Wroblewski, 1987] calls it a "computational sink". Copying is expensive in two ways: 1) it takes

*Visiting Research Scientist. Local email address: tomabech%atr-la.atr.co.jp@uunet.UU.NET

[1]The three parsing systems are based on: 1. Earley's algorithm, 2. active chart parsing, 3. generalized LR parsing.

[2]Please refer to [Kogure, 1989] for trivial time modification of Wroblewski's algorithm to handle loops.

[3]We estimate over 80% of unifications to be failures in our large-scale speech-to-speech translation system under development.

[4]Of course, whether that will improve the overall performance is another question.

time; 2) it takes space. Copying takes time essentially because the area in the random access memory needs to be dynamically allocated which is an expensive operation. [Godden, 1990] calculates the computation time cost of copying to be about 67 % of total parsing time in his TIME parsing system. This time/space burden of copying is non-trivial when we consider the fact that creation of unnecessary copies will eventually trigger garbage collections more often (in a Lisp environment) which will also slow down the overall performance of the parsing system. In general, parsing systems are always short of memory space (such as large LR tables of Tomita-LR parsers and expanding tables and charts of Earley and active chart parsers[5]), and the marginal addition or subtraction of the amount of memory space consumed by other parts of the system often has critical effects on the performance of these systems.

Considering the aforementioned problems, we propose the following principles to be the desirable conditions for a fast graph unification algorithm:

- **Copying should be performed only for successful unifications.**
- **Unification failures should be found as soon as possible.**

By way of definition we would like to categorize excessive copying of dags into Over Copying and Early Copying. Our definition of over copying is the same as Wroblewski's; however, our definition of early copying is slightly different.

- **Over Copying:** Two dags are created in order to create one new dag. – This typically happens when copies of two input dags are created prior to a destructive unification operation to build one new dag. ([Godden, 1990] calls such a unification: Eager Unification.). When two arcs point to the same node, over copying is often unavoidable with incremental copying schemes.
- **Early Copying:** Copies are created prior to the failure of unification so that copies created since the beginning of the unification up to the point of failure are wasted.

Wroblewski defines Early Copying as follows: "The argument dags are copied *before* unification started. If the unification fails then some of the copying is wasted effort" and restricts early copying to cases that only apply to copies that are created prior to a unification. Restricting early copying to copies that are made prior to a unification leaves a number of wasted copies that are created during a unification up to the point of failure to be uncovered by either of the above definitions for excessive copying. We would like Early Copying to mean all copies that are wasted due to a unification failure whether these copies are created before or during the actual unification operations.

Incremental copying has been accepted as an effective method of minimizing over copying and eliminating early copying as defined by Wroblewski. However, while being effective in minimizing over copying (it over copies only in some cases of convergent arcs into one node), incremental copying is ineffective in eliminating early copying as we define it.[6] Incremental copying is ineffective in eliminating early copying because when a graph unification algorithm recurses for shared arcs (i.e. the arcs with labels that exist in both input graphs), each created unification operation recursing into each shared arc is independent of other recursive calls into other arcs. In other words, the recursive calls into shared arcs are non-deterministic and there is no way for one particular recursion into a shared arc to know the result of future recursions into other shared arcs. Thus even if a particular recursion into one arc succeeds (with minimum over copying and no early copying in Wroblewski's sense), other arcs may eventually fail and thus the copies that are created in the successful arcs are all wasted. We consider it a drawback of incremental copying schemes that copies that are incrementally created up to the point of failure get wasted. This problem will be particularly felt when we consider parallel implementations of incremental copying algorithms. Because each recursion into shared arcs is non-deterministic, parallel processes can be created to work concurrently on all arcs. In each of the parallelly created processes for each shared arc, another recursion may take place creating more parallel processes. While some parallel recursive call into some arc may take time (due to a large number of subarcs, etc.) another non-deterministic call to other arcs may proceed deeper and deeper creating a large number of parallel processes. In the meantime, copies are incrementally created at different depths of subgraphs as long as the subgraphs of each of them are unified successfully. This way, when a failure is finally detected at some deep location in some subgraph, other numerous processes may have created a large number of copies that are wasted. Thus, early copying will be a significant problem when we consider parallelization of incremental copying unification algorithms.

2. Our Scheme

We would like to introduce an algorithm which addresses the criteria for fast unification discussed in the previous sections. It also handles loops without over copying (without any additional schemes such as those introduced by [Kogure, 1989]).

[5] For example, our phoneme-based generalized LR parser for speech input is always running on a swapping space because the LR table is too big.

[6] 'Early copying' will henceforth be used to refer to early copying as defined by us.

As a data structure, a node is represented with eight fields: type, arc-list, comp-arc-list, forward, copy, comp-arc-mark, forward-mark, and copy-mark. Although this number may seem high for a graph node data structure, the amount of memory consumed is not significantly different from that consumed by other algorithms. Type can be represented by three bits; comp-arc-mark, forward-mark, and copy-mark can be represented by short integers (i.e. fixnums); and comp-arc-list (just like arc-list) is a mere collection of pointers to memory locations. Thus this additional information is trivial in terms of memory cells consumed and because of this data structure the unification algorithm itself can remain simple.

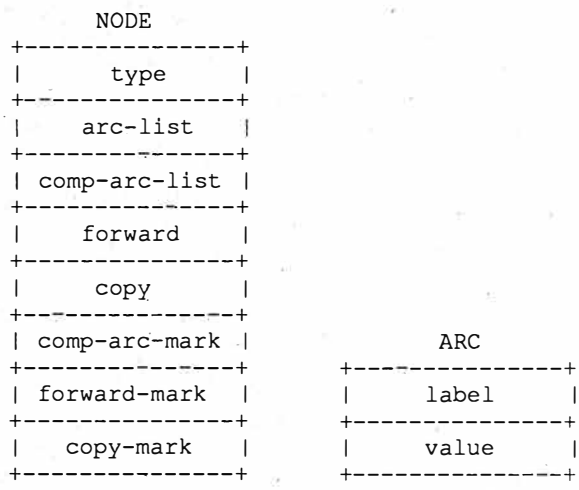


Figure 1: **Node and Arc Structures**

The representation for an arc is no different from that of other unification algorithms. Each arc has two fields for 'label' and 'value'. 'Label' is an atomic symbol which labels the arc, and 'value' is a pointer to a node.

The central notion of our algorithm is the dependency of the representational content on the global timing clock (or the global counter for the current generation of unification algorithms). This scheme was used in [Wroblewski, 1987] to invalidate the copy field of a node after one unification by incrementing a global counter. This is an extremely cheap operation but has the power to invalidate the copy fields of all nodes in the system simultaneously. In our algorithm, this dependency of the content of fields on global timing is adopted for arc lists, forwarding pointers, and copy pointers. Thus any modification made, such as adding forwarding links, copy links or arcs during one top-level unification (unify0) to any node in memory can be invalidated by one increment operation on the global timing counter. During unification (in unify1) and copying after a successful unification, the global timing ID for a specific field can be checked by comparing the content of mark fields with the global counter value and if they match then the content is respected, if not it is simply ignored. Thus the whole operation is a trivial addition to the original destructive unification algorithm (Pereira's and Wroblewski's unify1).

We have two kinds of arc lists 1) arc-list and comp-arc-list. Arc-list contains the arcs that are permanent (i.e., usual graph arcs) and comp-arc-list contains arcs that are only valid during one graph unification operation. We also have two kinds of forwarding links, i.e., permanent and temporary. A permanent forwarding link is the usual forwarding link found in other algorithms ([Pereira, 1985], [Wroblewski, 1987], etc). Temporary forwarding links are links that are only valid during one unification. The currency of the temporary links is determined by matching the content of the mark field for the links with the global counter and if they match then the content of this field is respected[7]. As in [Pereira, 1985], we have three types of nodes: 1) :atomic, 2) :bottom[8], and 3) :complex. :atomic type nodes represent atomic symbol values (such as Noun), :bottom type nodes are variables and :complex type nodes are nodes that have arcs coming out of them. Arcs are stored in the arc-list field. The atomic value is also stored in the arc-list if the node type is :atomic. :bottom nodes succeed in unifying with any nodes and the result of unification takes the type and the value of the node that the :bottom node was unified with. :atomic nodes succeed in unifying with :bottom nodes or :atomic nodes with the same value (stored in the arc-list). Unification of an :atomic node with a :complex node immediately fails. :complex nodes succeed in unifying with :bottom nodes or with :complex nodes whose subgraphs all unify. Arc values are always nodes and never symbolic values because the :atomic and :bottom nodes may be pointed to by multiple arcs (just as in structure sharing of :complex nodes) depending on grammar constraints, and we do not want arcs to contain terminal atomic values.

Below is our algorithm:

```
function UNIFY-DAG (dag1,dag2);    ;; toplevel
 RESULT :== catch with tag 'UNIFY-FAIL
            calling UNIFY0(dag1,dag2)
 increment *unify-global-counter*  ;; starts from 10
  return RESULT;
 end;

function UNIFY0 (dag1,dag2);
  if '*T* ==  UNIFY1(dag1,dag2);
      then COPY :== COPY-DAG-WITH-COMP-ARCS(dag1);
```

[7]In terms of forwarding links, we do not have a separate field for temporary forwarding links; instead, we designate the integer value 9 to represent a permanent forwarding link. We start incrementing the global counter from 10 so whenever the forward-mark is not 9 the integer value must equal the global counter value to respect the forwarding link.

[8]Bottom is called leaf in Pereira's algorithm.

```
          return COPY;
end;

function UNIFY1 (dag1-underef,dag2-underef);
 DAG1 :== DEREFERENCE-DAG(dag1-underef);
 DAG2 DAG(dag2-underef);

 if (DAG1 == DAG2)    ;;; i.e., 'eq' relation
    then return '*T*;
 else if (DAG1.type == :bottom) ;; variable
         then FORWARD-DAG(DAG1,DAG2,:temporary);
              return '*T*;
 else if (DAG2.type == :bottom)
         then FORWARD-DAG(DAG2,DAG1,:temporary);
              return '*T*;
 else if (DAG1.type == :atomic and
          DAG2.type == :atomic)
       then
        if (DAG1.arc-list == DAG2.arc-list)
              ;;;contains atomic values
         then FORWARD-DAG(DAG2,DAG1,
                                :temporary);
              return '*T*;
        else throw with keyword 'UNIFY-FAIL;
             ;;; return directly to unify-dag
                (throw/catch construct)
 else if (   DAG1.type == :atomic
          or DAG2.type == :atomic)
       then throw with keyword 'UNIFY-FAIL;
 else  NEW :== COMPLEMENTARCS(DAG2,DAG1);
       SHARED :== INTERSECTARCS(DAG1,DAG2);
       for each ARC in SHARED do
        RESULT :== UNIFY1(destination of the
                    shared arc for dag1,
                          destination of the
                    shared arc for dag2);
          if (RESULT =/= '*T*)
             throw with keyword 'UNIFY-FAIL;
        If (the recursive calls to UNIFY1
            successfully returned for all
            shared arcs)
      ;;; this check is actually unnecessary
        then
           FORWARD-DAG(DAG2,DAG1,:temporary);
           DAG1.comp-arc-mark :==
                      *unify-global-counter*;
           DAG1.comp-arc-list :== NEW
           return '*T*;
 end;

function COPY-DAG-WITH-COMP-ARCS(dag-underef);
 DAG :== DEREFERENCE-DAG(dag-underef);
 if (DAG.copy is non-empty
       and
    DAG.copy-mark == *unify-global-counter*)
   then return the content of DAG.copy;
                  ;;; i.e. existing copy
 else if (DAG.type == :atomic)
         COPY :== CREATE-NODE();
         COPY.type :== :atomic;
         COPY.arc-list :== DAG.arc-list;
              ;;; this is an atomic value
         DAG.copy :== COPY;
         DAG.copy-mark
                :== *unify-global-counter*;
         return COPY;
 else if (DAG.type == :bottom)
         COPY :== CREATE-NODE();
         COPY.type :== :bottom;
         DAG.copy :== COPY;
         DAG.copy-mark
                :== *unify-global-counter*;
         return COPY;
 else    COPY :== CREATENODE();
         COPY.type :== :complex;
         for all ARC in DAG.arc-list do
            NEWARC :== COPY-ARC-AND-COMP-ARC(ARC);
            push NEWARC into COPY.arc-list;
         if (DAG.comp-arc-list is non-empty
                    and
             DAG.comp-arc-mark ==
                      *unify-global-counter*)
            then
              for all COMP-ARC in
                       DAG.comp-arc-list do
               NEWARC :==
                 COPY-ARC-AND-COMP-ARC(COMP-ARC);
               push NEWARC into COPY.arc-list;
         DAG.copy :== COPY
         DAG.copy-mark :== *unify-global-counter*;
         return COPY;
  end;

function COPY-ARC-AND-COMP-ARC (input-arc)
 LABEL :== label of input-arc;
 VALUE :== COPY-DAG-WITH-COMP-ARCS
                      (value of input-arc);
 return a new arc with LABEL and VALUE;
 end;
```

The functions Complementarcs(dag1,dag2) and Intersectarcs(dag1,dag2) are the same as in Wroblewski's algorithm and return the set-difference (the arcs with labels that exist in dag1 but not in dag2) and intersection (the arcs with labels that exist both in dag1 and dag2) respectively. Dereference-dag(dag) recursively traverses the forwarding link to return the forwarded node. In doing so, it checks the forward-mark of the node and if the forward-mark value is 9 (9 represents a permanent forwarding link) or its value matches the current value of *unify-global-counter*, then the function returns the forwarded node; otherwise it simply returns the input node. Forward(dag1, dag2, :forward-type) puts (the pointer to) dag2 in the forward field of dag1. If the keyword in the function call is :temporary, the current value of the *unify-global-counter* is written in the forward-mark field of dag1. If the keyword is :permanent, 9 is written in the forward-mark field of dag1. Our algorithm itself does not require any permanent forwarding; however, the functionality is added because the grammar reader module that reads the path equation specifications into dag feature-structures uses permanent forwarding to merge the additional grammatical specifications into a graph structure[9]. The tem-

[9]We have been using Wroblewski's algorithm for the unification part of the parser and thus usage of (permanent)

porary forwarding links are necessary to handle reentrancy and loops. As soon as unification (at any level of recursion through shared arcs) succeeds, a temporary forwarding link is made from dag2 to dag1 (dag1 to dag2 if dag1 is of type :bottom). Thus, during unification, a node already unified by other recursive calls to unify1 within the same unify0 call has a temporary forwarding link from dag2 to dag1 (or dag1 to dag2). As a result, if this node becomes an input argument node, dereferencing the node causes dag1 and dag2 to become the same node and unification immediately succeeds. Thus a subgraph below an already unified node will not be checked more than once even if an argument graph has a loop. Also, during copying done subsequently to a successful unification, two arcs converging into the same node will not cause over copying simply because if a node already has a copy then the copy is returned. For example, as a case that may cause over copies in other schemes for dag2 convergent arcs, let us consider the case when the destination node has a corresponding node in dag1 and only one of the convergent arcs has a corresponding arc in dag1. This destination node is already temporarily forwarded to the node in dag1 (since the unification check was successful prior to copying). Once a copy is created for the corresponding dag1 node and recorded in the copy field of dag1, every time a convergent arc in dag2 that needs to be copied points to its destination node, dereferencing the node returns the corresponding node in dag1 and since a copy of it already exists, this copy is returned. Thus no duplicate copy is created[10].

As we just saw, the algorithm itself is simple. The basic control structure of the unification is similar to Pereira's and Wroblewski's unify1. The essential difference between our unify1 and the previous ones is that our unify1 is non-destructive. It is because the complementarcs(dag2,dag1) are added to the comp-arc-list of dag1 and not into the arc-list of dag1. Thus, as soon as we increment the global counter, the changes made to dag1 (i.e., addition of complement arcs into comp-arc-list) vanish. As long as the comp-arc-mark value matches that of the global counter the content of the comp-arc-list can be considered a part of arc-list and therefore, dag1 is the result of unification. Hence the name quasi-destructive graph unification. In order to create a copy for subsequent use we only need to make a copy of dag1 before we increment the global counter while respecting the content of the comp-arc-list of dag1.

Thus instead of calling other unification functions (such as unify2 of Wroblewski) for incrementally creating a copy node during a unification, we only need to create a copy after unification. Thus, if unification fails no copies are made at all (as in [Karttunen, 1986]'s scheme). Because unification that recurses into shared arcs carries no burden of incremental copying (i.e., it simply checks if nodes are compatible), as the depth of unification increases (i.e., the graph gets larger) the speed-up of our method should get conspicuous if a unification eventually fails. If all unifications during a parse are going to be successful, our algorithm should be as fast as or slightly slower than Wroblewski's algorithm[11]. Since a parse that does not fail on a single unification is unrealistic, the gain from our scheme should depend on the amount of unification failures that occur during a unification. As the number of failures per parse increases and the graphs that failed get larger, the speed-up from our algorithm should become more apparent. Therefore, the characteristics of our algorithm seem desirable. In the next section, we will see the actual results of experiments which compare our unification algorithm to Wroblewski's algorithm (slightly modified to handle variables and loops that are required by our HPSG based grammar).

3. Experiments

'Unifs' represents the total number of unifications during a parse (the number of calls to the top-level 'unify-dag', and not 'unify1'). 'USrate' represents the ratio of successful unifications to the total number of unifications. We parsed each sentence three times on a Symbolics 3620 using both unification methods and took the shortest elapsed time for both methods ('T' represents our scheme, 'W' represents Wroblewski's algorithm with a modification to handle loops and variables[12]). Data structures are the same for both

forwarding links is used by the grammar reader module.

[10]Copying of dag2 arcs happens for arcs that exist in dag2 but not in dag1 (i.e., Complementarcs(dag2,dag1)). Such arcs are pushed to the comp-arc-list of dag1 during unify1 and are copied into the arc-list of the copy during subsequent copying. If there is a loop or a convergence in arcs in dag1 or in arcs in dag2 that do not have corresponding arcs in dag1, then the mechanism is even simpler than the one discussed here. A copy is made once, and the same copy is simply returned every time another convergent arc points to the original node. It is because arcs are copied only from either dag1 or dag2.

[11]It may be slightly slower, because our unification recurses twice on a graph: once to unify and once to copy, whereas in incremental unification schemes copying is performed during the same recursion as unifying. Additional bookkeeping for incremental copying during unify2 may slightly offset this, however.

[12]Loops can be handled in Wroblewski's algorithm by checking whether an arc with the same label already exists when arcs are added to a node. And if such an arc already exists, we destructively unify the node which is the destination of the existing arc with the node which is the destination of the arc being added. If such an arc does not exist, we simply add the arc. ([Kogure, 1989]). Thus, loops can be handled very cheaply in Wroblewski's algorithm. Handling variables in Wroblewski's algorithm is basically the same as in our algorithm (i.e., Pereira's scheme), and the addition of

sent#	Unifs	USrate	Elapsed time(sec)		Num of Copies		Num of Conses	
			T	W	T	W	T	W
1	6	0.5	1.066	1.113	85	107	1231	1451
2	101	0.35	1.897	2.899	1418	2285	15166	23836
3	24	0.33	1.206	1.290	129	220	1734	2644
4	71	0.41	3.349	4.102	1635	2151	17133	22943
5	305	0.39	12.151	17.309	5529	9092	57405	93035
6	59	0.38	1.254	1.601	608	997	6873	10763
7	6	0.38	1.016	1.030	85	107	1175	1395
8	81	0.39	3.499	4.452	1780	2406	18718	24978
9	480	0.38	18.402	34.653	9466	15756	96985	167211
10	555	0.39	26.933	47.224	11789	18822	119629	189997
11	109	0.40	4.592	5.433	2047	2913	21871	30531
12	428	0.38	13.728	24.350	7933	13363	81536	135808
13	559	0.38	15.480	42.357	9976	17741	102489	180169
14	52	0.38	1.977	2.410	745	941	8272	10292
15	77	0.39	3.574	4.688	1590	2137	16946	22416
16	77	0.39	3.658	4.431	1590	2137	16943	22413

Figure 2: Comparison of our algorithm with Wroblewski's

unification algorithms (except for additional fields for a node in our algorithm, i.e., comp-arc-list, comp-arc-mark, and forward-mark). Same functions are used to interface with Earley's parser and the same subfunctions are used wherever possible (such as creation and access of arcs) to minimize the differences that are not purely algorithmic. 'Number of copies' represents the number of nodes created during each parse (and does not include the number of arc structures that are created during a parse). 'Number of conses' represents the amount of structure words consed during a parse. This number represents the real comparison of the amount of space being consumed by each unification algorithm (including added fields for nodes in our algorithm and arcs that are created in both algorithms).

We used Earley's parsing algorithm for the experiment. The Japanese grammar is based on HPSG analysis ([Pollard and Sag, 1987]) covering phenomena such as coordination, case adjunction, adjuncts, control, slash categories, zero-pronouns, interrogatives, WH constructs, and some pragmatics (speaker, hearer relations, politeness, etc.) ([Yoshimoto and Kogure, 1989]). The grammar covers many of the important linguistic phenomena in conversational Japanese. The grammar graphs which are converted from the path equations contain 2324 nodes. We used 16 sentences from a sample telephone conversation dialog which range from very short sentences (one word, i.e., *iie* 'no') to relatively long ones (such as *soredehakochirakarasochiranitourokuyoushiwoookuriitashimasu* 'In that case, we [speaker] will send you [hearer] the registration form.'). Thus, the number of unifications per sentence varied widely (from 6 to over 500).

this functionality can be ignored in terms of comparison to our algorithm. Our algorithm does not require any additional scheme to handle loops in input dags.

4. Discussion:

4.1. Comparison to Other Approaches

The control structure of our algorithm is identical to that of [Pereira, 1985]. However, instead of storing changes to the argument dags in the environment we store the changes in the dags themselves non-destructively. Because we do not use the environment, the log(d) overhead (where d is the number of nodes in a dag) associated with Pereira's scheme that is required during node access (to assemble the whole dag from the skeleton and the updates in the environment) is avoided in our scheme. We share the principle of storing changes in a restorable way with [Karttunen, 1986]'s reversible unification and copy graphs only after a successful unification. Karttunen originally introduced this scheme in order to replace the less efficient structure-sharing implementations ([Pereira, 1985], [Karttunen and Kay, 1985]). In Karttunen's method[13], whenever a destructive change is about to be made, the attribute value pairs[14] stored in the body of the node are saved into an array. The dag node structure itself is also saved in another array. These values are restored after the top level unification is completed. (A copy is made prior to the restoration operation if the unification was a successful one.) The difference between Karttunen's method and ours is that in our algorithm, one increment to the global counter can invalidate all the changes made to nodes, while in Karttunen's algorithm each node in the entire argument graph that has been destructively modified must be restored sep-

[13]The discussion of Kartunnen's method is based on the D-PATR implementation on Xerox machines ([Karttunen, 1986]).

[14]I.e., arc structures: 'label' and 'value' pairs in our vocabulary.

arately by retrieving the attribute-values saved in an array and resetting the values into the dag structure skeltons saved in another array. In both Karttunen's and our algorithm, there will be a non-destructive (reversible, and quasi-destructive) saving of intersection arcs that may be wasted when a subgraph of a particular node successfully unifies but the final unification fails due to a failure in some other part of the argument graphs. This is not a problem in our method because the temporary change made to a node is performed as pushing pointers into already existing structures (nodes) and it does not require entirely new structures to be created and dynamically allocated memory (which was necessary for the copy (create-node) operation).[15] [Godden, 1990] presents a method of using lazy evaluation in unification which seems to be one successful actualization of [Karttunen and Kay, 1985]'s lazy evaluation idea. One question about lazy evaluation is that the efficiency of lazy evaluation varies depending upon the particular hardware and programming language environment. For example, in CommonLisp, to attain a lazy evaluation, as soon as a function is delayed, a closure (or a structure) needs to be created receiving a dynamic allocation of memory (just as in creating a copy node). Thus, there is a shift of memory and associated computation consumed from making copies to making closures. In terms of memory cells saved, although the lazy scheme may reduce the total number of copies created, if we consider the memory consumed to create closures, the saving may be significantly canceled. In terms of speed, since delayed evaluation requires additional bookkeeping, how schemes such as the one introduced by [Godden, 1990] would compare with non-lazy incremental copying schemes is an open question. Unfortunately Godden offers a comparison of his algorithm with one that uses a full copying method (i.e. his Eager Copying) which is already significantly slower than Wroblewski's algorithm. However, no comparison is offered with prevailing unification schemes such as Wroblewski's. With the complexity for lazy evaluation and the memory consumed for delayed closures added, it is hard to estimate whether lazy unification runs considerably faster than Wroblewski's incremental copying scheme.

Finally, when we consider parallelization of unification algorithms, it seems that the quasi-destructive unification scheme is more suitable for parallelization than the past methods. When we parallelize graph unification, the concurrent recursive calls into shared arcs should be the element contributing to the speed up. On the other hand, that may require synchronization between parallel recursive processes which in turn may undermine the speed up element due to parallelization. Also, concurrently accessing shared data (i.e., global variables, etc.) causes lock/unlock synchronization on the global memory location and that also undermines the effect of parallelization. These two problems seem particularly applicable to incremental copying schemes (such as [Wroblewski, 1987] and [Godden, 1990]) because there may be multiple simultaneous write operations on a copy when recursive calls to the shared arcs at each level return successfully. Our algorithm does not suffer from this simultaneous write lock/unlock problem because there will be no write operation to a node during unification checks (i.e., no writing is performed until the unification of entire argument dags actually succeeds[16]).

In terms of simultaneous writes to shared global variables, Both structure sharing schemes and the reversible unification seem vulnerable to this problem because values are stored into global data and the concurrent processes must lock and unlock these global locations every time they access the data. For example, Kartunnen's reversible unification scheme requires two global arrays to store the original feature-value pairs and the dag node cells. When parallel recursive unification calls into shared arcs are performed and node values are saved into the arrays concurrently, the processes need to be queued (lock/unlock synchronization) to access the arrays[17]. The same problem will be caused during writes to 'copying environments' in the lazy unification scheme. Our algorithm does not suffer from simultaneous writes to global shared variable simply because 1) no saving is performed at all 2) changes are local. Instead of saving original values, changes are recorded distributedly (locally) into each node that

[15] Although, in Karttunen's method it may become rather expensive if the arrays require resizing during the saving operation of the subgraphs. This is another characteristic of Kartunnen's method that two arrays need to be originally allocated memory. If the allocated arrays are too big then we will be wasting the unused cells, if it is too small, then there will be array resizing operations during unification which can be costly. Because amount of destructive operations during unifications vary significantly sentence to sentence, determining the ideal initial array size for Kartunnen's method is not trivial.

[16] In our current parallel implementation ([Tomabechi and Fujioka, ms]), the quasi-destructive addition of intersection arcs to a node does not occur until all parallel recursive calls into subgraphs succeed. This can be performed without any harm because 1) any addition to the comp-arc-list is harmless until actual copying is performed after a successful unification; 2) additions to comp-arc-list are performed only once per node and therefore, this will not cause the lock/unlock problem due to multiple simultaneous write operations. However, the addition of temporary forwarding links needs to wait until the top-level unification successfully returns.

[17] Depending on parallel machine architectures and operating system implementations, simultaneous read/read and read/write may not be problems, however, simultaneous write/write is normally inherently problematic and needs to be synchronized. Simultaneous write/write into save arrays is inevitable if we parallelize Kartunnen's scheme because writing to arrays (i.e., both feature-value pair array and the dag cell array) must occur during the save operation.

is being quasi-destructively modified. Therefore, there will be no global shared data associated with the saving of original dag values. Changes are simply nullified by the increment on the global counter and therefore no saving operation is necessary. Overall, we have seen in our experiments (reported in [Tomabechi and Fujioka, ms]) that our algorithm recorded about 75 percent of effective parallelization rate (meaning that the 75 percent of unifications into shared arcs were parallelly performed both horizontally and vertically) ([Tomabechi and Fujioka, ms][18]).

5. Conclusion

The algorithm introduced in this paper runs significantly faster than Wroblewski's algorithm using Earley's parser and an HPSG based grammar developed at ATR. The gain comes from the fact that our algorithm does not create any over copies or early copies. In Wroblewski's algorithm, although over copies are essentially avoided, early copies (by our definition) are a significant problem because about 60 percent of unifications result in failure in a successful parse in our sample parses. The additional set-difference operation required for incremental copying during unify2 may also be contributing to the slower speed of Wroblewski's algorithm. Given that our sample grammar is relatively small, we would expect that the difference in the performance between the incremental copying schemes and ours will expand as the grammar size increases and both the number of failures[19] and the size of the wasted subgraphs of failed unifications become larger. Since our algorithm is essentially parallel, parallelization is one logical choice to pursue further speedup. Parallel processes can be continuously created as unify1 recurses deeper and deeper without creating any copies by simply looking for a possible failure of the unification (and preparing for successive copying in case unification succeeds). So far, we have completed a preliminary implementation on a shared memory parallel hardware with about 75 percent of effective parallelization rate. With the simplicity of our algorithm and the ease of implementing it (compared to both incremental copying schemes and lazy schemes), combined with the demonstrated speed of the algorithm, the algorithm could be a viable alternative to existing unification algorithms used in the existing parsing schemes as well as a part of future parsing systems.

[18] Please refer to this paper for detail of parallel quasi-destructive unification algorithm and experiments using the algorithm.

[19] For example, in our large-scale speech-to-speech translation system under development, the USrate is estimated to be under 20%, i.e., over 80% of unifications are estimated to be failures.

ACKNOWLEDGMENTS

The author would like to thank Akira Kurematsu, Tsuyoshi Morimoto, Hitoshi Iida, Osamu Furuse, Masaaki Nagata, Toshiyuki Takezawa and other members of ATR. Thanks are also due to Margalit Zabludowski for comments on the final version of this paper and Takako Fujioka for assistance in implementing the parallel version of our algorithm.

Appendix: Implementation

The unification algorithms, Earley parser and the HPSG path equation to graph converter programs are implemented in Common Lisp on a Symbolics machine. The preliminary parallel version of our unification algorithm is currently implemented on a Sequent Symmetry closely coupled shared-memory parallel machine with 15 CPUs running Allegro CLiP parallel CommonLisp based on a micro-tasking parallelism using light-weight processes.

References

[Godden, 1990] Godden, K. "Lazy Unification" In *Proceedings of ACL-90*. 1990.

[Karttunen, 1986] Karttunen, L. *Development Environment for Unification-Based Grammars*. Report CSLI-86-61. Center for the Study of Language and Information, 1986.

[Karttunen, 1986] Karttunen, L. "D-PATR: A Development Environment for Unification-Based Grammars". In *Proceedings of COLING-86*. 1986.

[Karttunen and Kay, 1985] Karttunen, L. and Kay, M. "Structure Sharing with Binary Trees". In *Proceedings of ACL-85*. 1985.

[Kasper, 1987] Kasper, R. "A Unification Method for Disjunctive Feature Descriptions". In *Proceedings of ACL-87*. 1987.

[Kogure, 1989] Kogure, K. *A Study on Feature Structures and Unification*. ATR Technical Report. TR-1-0032. 1988.

[Pereira, 1985] Pereira, F. "A Structure-Sharing Representation for Unification-Based Grammar Formalisms". In *Proceedings of ACL-85*. 1985.

[Pollard and Sag, 1987] Pollard, C. and Sag, A. *Information-based Syntax and Semantics*. Vol 1, CSLI, 1987.

[Tomabechi and Fujioka, ms] *Parallel Quasi-Destructive Graph Unification*. Manuscript (in print as ATR Technical Report).

[Yoshimoto and Kogure, 1989] Yoshimoto, K. and Kogure, K. *Japanese Sentence Analysis by means of Phrase Structure Grammar*. ATR Technical Report. TR-1-0049. 1989.

[Wroblewski, 1987] Wroblewski, D."Nondestructive Graph Unification" In *Proceedings of AAAI87*. 1987.

Unification Algorithms for Massively Parallel Computers*

Hiroaki Kitano
Center for Machine Translation
Carnegie Mellon University
Pittsburgh, PA 15213 U.S.A.
hiroaki@cs.cmu.edu

NEC Corporation
2-11-5 Shibaura, Minato-ku
Tokyo, 108 Japan

ABSTRACT

This paper describes unification algorithms for fine-grained massively parallel computers. The algorithms are based on a parallel marker-passing scheme. The marker-passing scheme in our algorithms carry only bit-vectors, address pointers and values. Because of their simplicity, our algorithms can be implemented on various architectures of massively parallel machines without loosing the inherent benefits of parallel computation. Also, we describe two augmentations of unification algorithms such as multiple unification and fuzzy unification. Experimental results indicate that our algorithm attaines more than 500 unification per seconds (for DAGs of average depth of 4) and has a linear time-complexity. This leads to possible implementations of massively parallel natural language parsing with full linguistic analysis.

1. Introduction

This paper describes unification algorithms using parallel marker-passing scheme. The purpose of this paper is to show parallel unification algorithms which are simple enough to be implemented by massively parallel machines, and have some novel features.

Unification is a basic operation in computational linguistics. However, this operation is known to be computationally expensive, and thus is considered a major bottleneck in improving the performance of natural language processing systems. A search for efficient algorithms has been conducted by many researchers involving parallel algorithms such as [Yasuura, 1984]. However, theoretical lower-bound was shown by [Dwork et. al., 1984] that unifiability is log-space complete for P. This leads to [Knight, 1989]'s conclusion that use of massively parallel machines will not significantly improve the speed of unification. Then, why do we propose a parallel unification? We have three major reasons.

First, although theoretical limitation for speed up has been shown for full unification, parallelization of unification actually improves performance of the entire system. This improvement of performance is a clear benefit for practical natural language processing systems, in particular for tasks like spoken language processing where real-time processing is essential. In addition, we propose parallel unification algorithms which attained a time-complexity of $o(D)$ where D is a depth of the deepest path in DAGs to be unified. We achieved this by assuming all disjunctions are pre-expanded into several DAGs so that each pair of DAGs does not contain disjunctions, and so that higher parallelism can be maintained through out the unification process. This is a reasonable assumption when we implement unification on massively parallel machines, where the basic implementation strategy is a memory-intensive approach allowing time-complexity to be converted into space-complexity. Thus, although we do not discover faster full unification with disjunction, we discovered a means to substantially speed up unification on the massively parallel machines.

Second, we designed our algorithm for massively parallel machines where each processor has relatively low processing capability. We only require each processing unit to have some basic operations and the capability to pass bit-markers, pointers to other processing units, and numeric values. This design decision aims at the accomplishment of two things — development of practical unification algorithms for massively parallel computers such as SNAP [Moldovan et. al., 1989] and Connection Machine [Hillis, 1985], and development of algorithms for specialized unification hardware such as unification chips or unification co-processors. Functionalities of massively parallel machines are severely limited due to the weak processing capability of each unit. Advantages of massively parallel machines for semantic processing, such as contextual priming, are widely recognized. However, in implementing serious natural language parsers, unification operation is essential. Unfortunately, we have not seen any algorithm which assumes low processing capability of each processor in massively parallel machines. Although some machines support high-level language, such as C or lisp, automatic parallelization does not guarantee efficiency of actual operations. Thus, designing unification algorithms for massively parallel machines has great impact on exploring maximum potential of these machines for natural language processing. One other reason is that, by assuming each processor has

*This work has been supported in part by the National Science Foundation under grant MIP-90/09109.

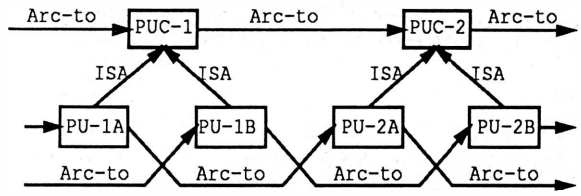


Figure 1: PU Class Nodes and PUs

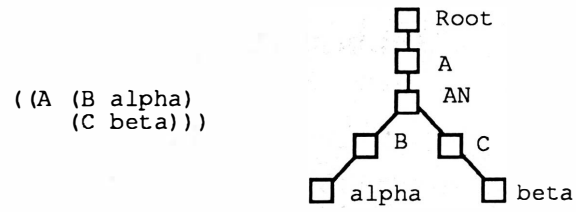


Figure 2: Representation of Nodes and Arcs

low computation power, our algorithms could be implementable as unification co-processor boards using numbers of less-powerful processors. A possibility for such a compact acceralator would be the clear benefit for the natural language community.

Third, our algorithms can easily entail some novel features such as multiple unification and fuzzy unification. These features have not been considered in past unification literature. It can also incorporate typed unification. Multiple unification is a unification between more than two trees or DAGs. Our algorithms enable this scheme without undermining its performance. Fuzzy unification allows unification of un-unifiable DAGs, but assigns a cost of violations. This would be useful for applications such as spoken language processing where handling of ungrammatical input is essential, because subtle ungrammaticalities can be overlooked.

2. Architecture, Representation and Notations

2.1. Architecture

We assume a parallel architecture where numbers of processing units are interconnected. The Processor Unit (PU) is a basic element of the system. It has its own processing capability and memory. This can be physical or logical, but, of course, we assume each unit is actually implemented as hardware. The Processor Unit Class (PUC) is a class of PUs which has several PUs as instances of the PUC. For each PUC, one PU is assigned to manage instances of the class. Figure 1 illustrates relations between PUCs and PUs. PUC-1 has instances PU-1A and PU-1B, and PUC-2 has instances PU-2A and PU-2B. This relation will be established when DAGs are loaded onto the unification co-processor.

We assume each PU's memory is is composed of a bit markers register, value register, and pointer memory for fan-in connections, fan-out connections, and address registers.

2.2. Represenation of Tree and DAGs

Trees or DAGs are represented as PUs and their connections. Each arc and node is assigned to each PU. Figure 2 shows how trees and DAGs are represented using PUs. In Figure 2, PUs are represented as square. Lines represent directed arcs. PUs in the middle of arcs represent labels of arcs. Each PU is connected by an Arc-to type link. When mapping feature structures on PUs, all PUs representing tree-0 or DAG-0 are marked with a marker 0, and all PUs representing tree-1 or DAG-1 are marked with a marker 1. PUs representing values have a marker V, and that of features have a marker F. Root PUs have a marker R.

2.3. Notations

The following notations will be used in describing algorithms:

PU(a,b,...,z): PU with specific markers set. PU(1,S,V) means that the PU has marker 1, S, and V. Negation can be used. For example, PU(1,S,-V) means PU has marker 1 and S set, but not V. Unspecified markers are don't care markers. Predicates can be used to specify conditions.

&PU(a,b,...,z): Address of PU which satisfies conditions specified.

Propagate: Propagation of markers through Arc-to link forward, i.e. direction from root to edge.

Back-Propagate: Propagation of markers through Arc-to link backward, i.e. direction from edge to root. This should not be confused with back-propagation in connectionist learning.

P-Address: Variable which can propagate or back-propagate an address of a PU.

The following instruction set will be used:

Propagate (Marker, Origin, Destination, Initial-action, Intermediate-action, Final-action): Propagate marker from origin to destination. Before propagation starts, do initial-action. At each PU during propagation, do intermediate-action, and at the destination PU, do final-action. In some special cases, destination is specified as 1. This means that markers are propagated only for one traverse.

Back-Propagate (Marker, Origin, Destination, Initial-action, Intermediate-action, Final-action): Back-propagation version of propagate instruction.

Mark(Marker,PU): Set marker to PUs. When PU is not specified (i.e. Mark(V)), the mark operation is performed to a current PU.

Set(Variable,Value): Set operator set a value specified in the second argument to the variable specified in the first argument. For example, Set(P-Address,&PU) sets an address of current PU to P-Address.

Connect(Arc-type,Origin,Destination): Create link of arc-type between origin and destination.

Other instructions such as **Create-Node(a,b,...,z), In(P-Address, From-Address), Equal(P-Address, &PU),** and **GLB-Search(...)** will be explained in sections where they are used. In some cases, if-then-else control sequence is used for ease of understanding. However, obviously, this can be implemented using logical bit-marker operations such as (AND 1 2 4) followed by a propagation instruction, such as Propagate(P-Address,PU(4),PU(V)....). This case, (AND 1 2 4) is a logical operation that set marker 4 when markers 1 and 2 exist. This instruction sequence should be read as: if there are PUs such that PU(1,2), then propagate(P-Address,PU(1,2),PU(V)....).

3. Pseudo-Unification

Pseudo-unification or tree-unification is a unification between trees [Tomita and Knight, 1988]. The advantage of using pseudo-unification, instead of full-unification (or graph-unification), is that it can be implemented easier (less resource requirements and a simpler algorithm) and faster than full-unification. Yet, practically, pseudo-unification can cover a substantial range of linguistic phenomena. Actually, KBMT-89 [Nirenberg et. al., 1989] (a knowledge-based machine translation system based on LFG, and developed at the Center for Machine Translation at Carnegie Mellon University) was implemented using pseudo-unification.

3.1. The Algorithm

The algorithm which we describe in this section accounts for all non-disjunctive cases of pseudo-unification. Tree-0 and Tree-1 are unified (figure 3). Our algorithm for destructive tree unification consists of three parts:

1. Shared Node Detection
2. Failure Detection
3. Merging

3.1.1. Shared Node Detection

The goal of the shared node detection stage, or the common feature detection stage, is to set S markers to all nodes that are shared between trees. Step 1 carry out this stage.

Figure 3(a) shows the initial state of trees loaded into a PU network. First of all, an address of a PUC of a root PU of the tree-0 is set to P-Address. Then, P-Address is propagated until it gets to a PU which has V marker set. During this propagation, Check-Shared is conducted at each PU which P-Address traverses through. &ISA(Root) returns an address of the PUC of the Root PU. By the same token, &ISA(PU-0) returns an address of the PUC of the PU-0. The result is shown in 3(b). All shared PUs are indicated by solid circles. Some important markers on each PU are shown in brackets, but some markers are ignored due to diagram space.

3.1.2. Failure Detection

Next, we would like to detect conflicts. We assume that if two different value units are linked to the PUs both under the same PUC, and the PU is a shared arc unit, then unification should fail. Step 2 and 3 carry out this stage.

Back-Propagate starts from terminal nodes which are not shared. The purpose of this back-propagation is to identify pre-terminal PUs which are Arcs. In case of Figure 3, tree-0 and tree-1 are unifiable.

3.1.3. Merging

Since unifiability is assured in the failure detection stage, all we need is to merge two trees. Step 4, 5, 6, and 7 carry out this stage.

Back-propagation is used to search PUs which unshared leaves should be connected to. Figure 3(c) indicate PUs involved in this process. Propagation starts from PU(1,V,-S) and goes up until it meets a PU which is shared. These PUs are places where unshared branches should be connected. Next, propagate an address of each PUs for one traverse. Now, relevant PUs have an address of PUs which should be connected. Connect a PU with markers P-Address, 0, and B and a PU with markers P-Address, 1, and T with Arc-to. Propagate marker 0 from PU with P-Address, 0, and B. As a result, we get a unified tree consisting of PUs marked with 0.

4. Full-Unification

Although pseudo-unification does quite a good job in most practical cases, there are cases where graph-unification is necessary. Lack of the re-entrance in the pseudo-unification forces grammar writers to subdivide their grammar rules to cope with various cases of re-entrance because re-entrant structure must be expanded to trees. This section presents full-unification (destructive version).

1: Propagate(P-Address, Root, PU(V), Set(P-Address,&ISA(Root)), Check-Shared, nil)
Check-Shared: If there is a PU (PU-1), under the same PUC, such that PU(1,In(P-Address, From-Addresses)), then Mark(S), Mark(S,PU-1), and Set(P-Address,&ISA(PU-0)), else abort propagation.
2: Back-Propagate(PT,PU(V,-S),1,nil,nil,Mark(PT))
3: If there is a PU such that PU(PT,S), then unification is a failure.
4: Back-Propagate(P-Address,PU(1,V,-S),PU(S), Set(P-Address, &PU(1,V,-S)), nil, Mark(B,PU(S,P-Address)))
5: Propagate(P-Address, PU(B), 1, Set(P-Address,&PU(B)), nil, Mark(T))
6: Connect(Arc-to, PU(P-Address,0,B), PU(P-Address,1,T))
7: Propagate(0, PU(0,B), PU(V), nil, Mark(0), Mark(0))

Table 1: Pseudo-Unification Algorithm

4.1. The Algorithm

In full-unification, we only need to add merging of arcs which is not covered in the pseudo-unification algorithm.

1. Shared Node Detection Stage
2. Failure Detection Stage
3. Merging Stage
4. Arc Merging Stage

DAG-0 and DAG-1 are unified (figure 4). In figure 4(a), shared nodes are detected and indicated by solid circles. Figure 4(b) and (c) shows the merging stage. In figure 4(b) top and bottom PUs are marked and then merged in figure 4(c). Up to this point, we can simply apply algorithms presented for pseudo-unification. However, in unifying DAGs, we must take into account the existence of unshared arcs which are in between shared PUs that are not handled in the merging stage in the pseudo-unification algorithm. An arc merging stage merges such arcs into the DAG. The algorithm presented here covers most of practical cases of nondisjunctive graph unification, but there are some cases which the algorithm does not provide correct result. However, even in such cases, a simple post-processing can modify the graph to provide correct results.

4.1.1. Arc Merging

The arc merging stage for the destructive graph unification is shown in table 2. For all nodes with marker F and 1, but not S, propagate marker E. Propagation stops when it arrives at a node marked S. Back-Propagate P-Address until it arrives at a node with S. For all nodes which have S and P-Address, mark B. Propagate marker B for one traverse, and mark destination node with T. Connect a node with markers P-Address, 0, and B and a node with markers P-Address, 1, and T with Arc-to. Propagate marker 0 from a node with P-Address, 0, and B.

5. Nondestructive Unification

So far we have been discussing destructive unification algorithms where represented feature structures are destroyed in the process of unification. Obviously, this would be problematic because (1) it destroys the original feature structure even when the feature structure needs to have its unifiability examined against more than one feature structure, and (2) destructive unification involves over-copying and early-copying [Wroblewski, 1988]

In this section, we further extend algorithms presented so far, and present a nondestructive graph unification algorithm. To implement the nondestructive graph unification, new nodes and arcs need to be created by assigning them on empty PUs. Instead of passing only P-Address, as we have been using so far, we pass P-Address and N-Address (an address of the newly assigned PU). Given two DAGs, the algorithm in table 3 creates a new DAG as a result of unification.

Figure 5, 6 and 7 show intermediate processes. DAG-0 and DAG-1 are unified and result in DAG-2. Figure 5 is a state after the shared node is detected. Solid circles indicate PUs for shared nodes. In figure 6, all unifiable branches of DAG-0 and DAG-1 are merged to DAG-2 to create New DAG-2. In figure 7 intermediate arcs are merged into DAG-2, and create Final DAG-2. One big difference between nondestructive graph unification and destructive unification is that, in nondestructive unification, new PUs are assigned when unifiable subgraphs from DAG-0 and DAG-1 are merged into DAG-2, whereas destructive unification is simply marked with 0 at the merging process. For this reason, **Append-New-Node** assigns a new PU for each node merged to DAG-2, and connects it to existing DAG-2 structure. Then, pointers to the merged PU in DAG-2 and an equivalent PU in DAG-0 or DAG-1, are propagated so that the next PU can be connected to them.

6. Typed Unification

The ψ-terms proposed in [Ait-Kaci, 1984] are similar to the feature structure, but the functor is retained. This provides a filter under unification because two feature structures with incompatible functors cannot be unified. When a conflict is detected, it is resolved by finding the greatest lower bound (GLB) of two items

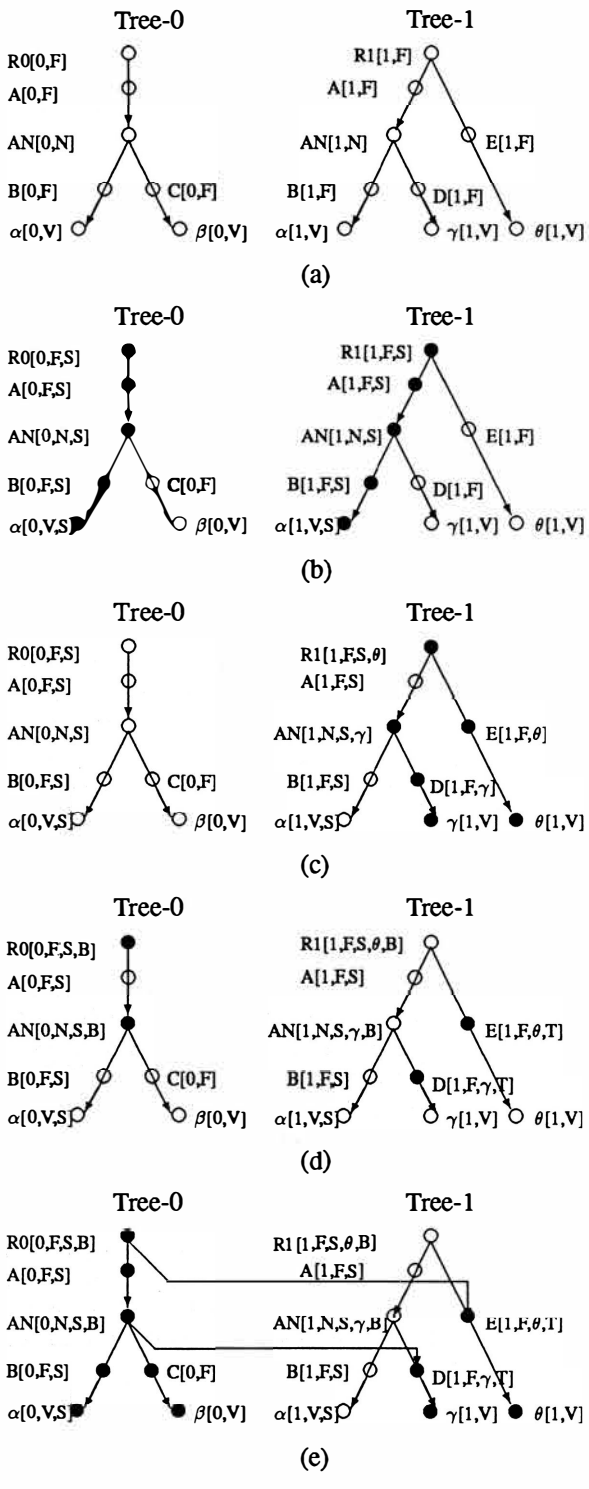


Figure 3: Pseudo-Unification

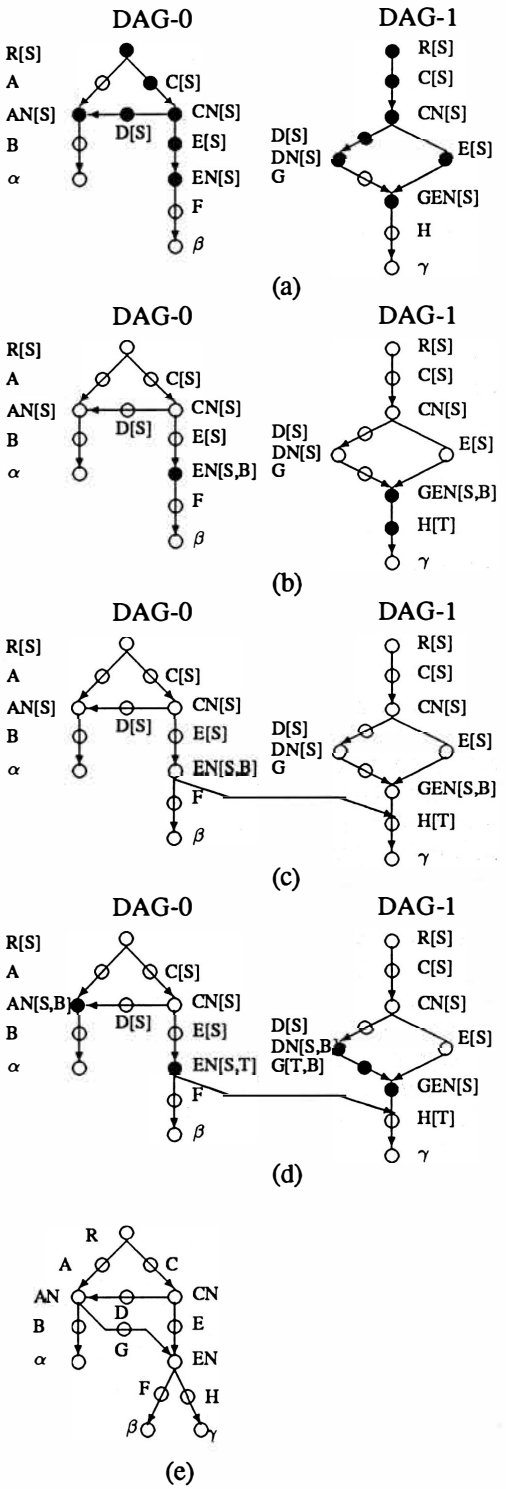


Figure 4: Graph Unification

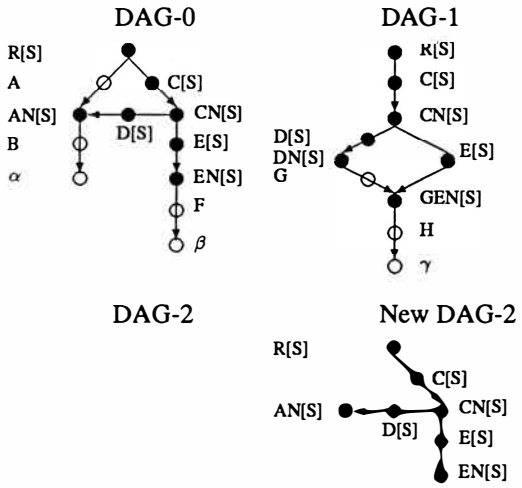


Figure 5: Nondestructive Graph Unification: Detect Shared Nodes

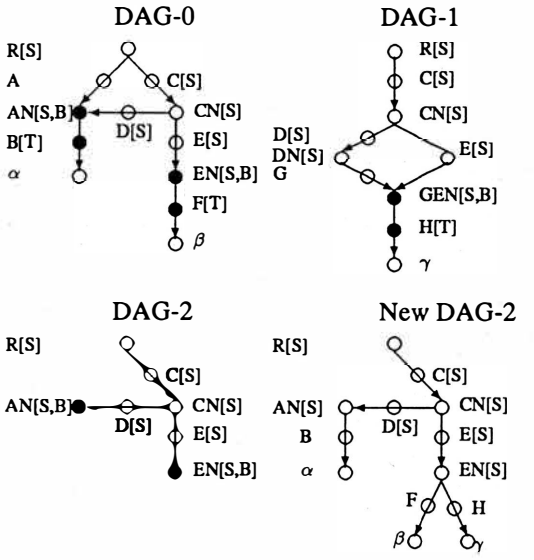


Figure 6: Nondestructive Graph Unification: Merge

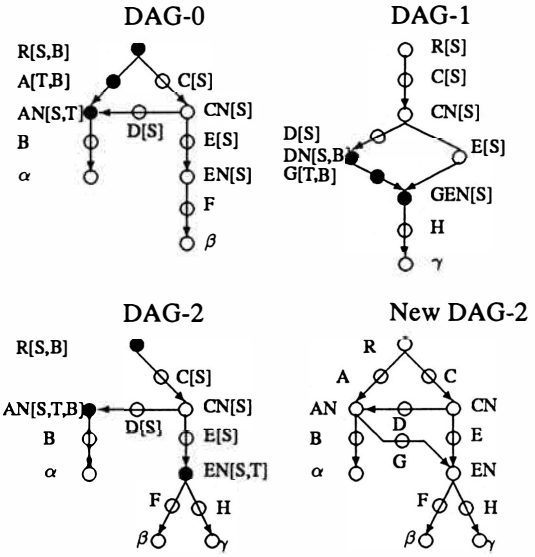


Figure 7: Nondestructive Graph Unification: Merge Internal Arcs

8: Propagate(E,PU(F,1,-S),PU(S),nil,nil,Mark(E))
9: Back-Propagate(P-Address, PU(E), PU(S), Set(P-Address,&PU), Set(P-Address,&PU), Mark(B))
10: Propagate(P-Address, PU(B), 1, Set(P-Address,&PU), nil, Mark(T))
11: Connect(Arc-to, PU(P-Address,0,B), PU(P-Address,1,T))
12: Propagate(0, PU(P-Address,0,B), PU(V), nil, Mark(0), Mark(0))

Table 2: Arc Merging Stage in Destructive Graph Unification

1: Propagate(P-Address N-Address, Root, PU(V), Set(P-Address,&ISA(Root)) Set(N-Address,&ISA(New-Root)), Check-Shared, nil)
Check-Shared: If there is a PU (PU-1), under the same PUC, such that PU(1,In(P-Address,From-Addresses)), then Mark(S), Mark(S,PU-1), Set(P-Address,&ISA(PU-0)), Create-Node(2,S,N-Address), Connect(Arc-to,PU(N-Address),PU(2,S,N-Address)), and Set(N-Address,&PU(2,S,N-Address)), else abort propagation.
2: Back-Propagate(PT,PU(V,-S),1,nil,nil,Mark(PT))
3: If there is a PU such that PU(PT,S), then unification is failure.
4: Back-Propagate(P-Address, PU(1,V,-S), PU(S), Set(P-Address,&PU(1,V,-S)), nil, Mark(B,PU(S,P-Address)))
5: Propagate(P-Address, PU(B), 1, Set(P-Address,&PU(B)), nil, Mark(T))
6: Propagate(P-Address N-Address, PU(B), PU(V), Set(P-Address,&PU(2,P-Address), Append-New-Nodes, nil)
Append-New-Nodes: If a cuurent PU is PU(0,-S) or PU(1,-S), then Create-Node(2,N-Address), Connect(Arc-to,PU(P-Address),PU(2,N-Address)), Set(N-Address,&PU(2,N-Address)), and Set(P-Address,&PU(2,P-Address)), else abort propagation.
7: Propagate(E, PU(F,1,-S), PU(S), nil, nil, Mark(E))
8: Back-Propagate(P-Address, PU(E), PU(S), Set(P-Address,&PU), Set(P-Address,&PU), Mark(B))
9: Propagate(P-Address, PU(B), 1, Set(P-Address,&PU), nil, Mark(T))
10: Propagate(P-Address N-Address, PU(B), PU(E), Set(P-Address,&PU(2,P-Address), Append-New-Nodes, Connect(Arc-to,N-Address,PU))

Table 3: Non-destructive Graph Unification Algorithm

in the taxonomic hierarchy. One way of implementing this scheme is to incorporate a search of hierarchy at the shared node detection. Perform the instructions shown in table 4 immediately after the shared node detection stage:

GLB-Search is a special instruction where propagation of markers start from nodes with V markers set but not S markers, and P-Address is propagated through ISA hierarchy downward. At each PU during the traversal, the current PU's address is set to GLB-Address, and it is propagated through ISA link upward. When GLB-Address arrives at a PU with V marker set but not S marker, it means there are GLB between the origin PU and the destination PU. Now, GLB-Search is conducted backwards, starting from the previous destination PU. This gives an address of the GLB PU to the originated PU. Thus, both PUs have an addres of the GLB PU. When one PU (PU-a) is under the other PU's (PU-b) ISA hierarchy, a GLB PU should be PU-a. Using the same mechanism, an address of PU-a is given to both PUs. However, this time GLB-Address propagation is not involved since GLB-PU itself is a destination PU. At the merging stage, PUs representing GLB should be merged instead of PUs in DAG-0 or DAG-1 (when GLB is one of the PU in DAG-0 or DAG-1, the PU in these DAGs can be merged). This can be done by using pointers to the GLB PUs propagated to PUs in DAG-0 and DAG-1. This mechanism enables typed unification.

7. Augmenting Unification

7.1. Fuzzy Unification

Traditionally, unification has been a logical operation, and thus, its failure resulted in hard rejection. We propose an alternative scheme called a *fuzzy unification* or a *soft rejection unification.* Contrary to the traditional unification which only returns nil when failed, a new unification scheme returns a partially unified feature structure and a value which indicates the degree of failure. In the soft rejection unification, each arc is assigned with a value which is accumulated when unification in its subgraph was failed. Meanings of the value can vary depending upon application and specific implementation. It can be a cost of violation or a probability measure of which violation will happen.

Unification operation with such property is significant for many applications which require robust parsing. For example, speech input processing requires integrated processing of a speech recognition module and linguistic parsing in order to limit the scope of search (reduce perplexity) which in turn improves recognition

Typed 1:	GLB-Search(P-Address GLB-Address, PU(V,-S), PU(V), Set(P-Address,&PU) Set(GLB-Address,&PU), Set(GLB-Address,&PU), nil)
Typed 2:	GLB-Search(P-Address GLB-Address, PU(V,-S,P-Address), PU(V), Set(P-Address,&PU) Set(GLB-Address,&PU), Set(GLB-Address,&PU), nil)
Typed 3:	Mark(S, PU(Equal(P-Address,&PU)))

Table 4: Type Checking in Typed Unification

rate. While spoken language inherently involves erroneous sentences, use of the traditional hard-rejection-type unification cannot be applied as it is — parsing needs to proceed even with minor syntactic failures. Some relaxation techniques have been proposed for detecting and overlooking minor errors by allowing some of the constraints to be ignored. However, traditional relaxation methods require multiple unification operations to check against sets of constraint equations, resulting in substantial overhead against conventional unification-based parsing. In addition, these relaxation methods did not assign weights or the probability that certain violations will happen. This would have adverse effects in reducing perplexity, because all possible errors are granted or predicted with equal weights. Since the likelihood of certain violations happening can be statistically obtained, providing *a priori* probability of such violations would help improve recognition rate.

For example, in a sentence *John want to attend the conference.* Although *John* and *want* cause violation in the third-person-present-singular constraint, we do not want that parse to be aborted since its semantics can be easily recovered in a post-processing. However, we want to add a cost to such parse so that if a speech recognition module provided two word hypotheses of *want* and *wants*, *John wants* ... would be selected as a most probable hypothesis.

This extension is trivial in our algorithm. The failure detection stage is revised as seen in table 5.

ADD-value adds values of markers at the root node. Alternatively, more sophisticated computation, instead of ADD, can be used to determine the degree of unifiability.

7.2. Multiple Unification

Traditionally, unification has been defined as an operation between between two DAGs; it takes two DAGs and returns a unified DAG or nil when failed. We extend this notion and propose multiple unification — unification of more than two DAGs. This extension would benefit processing of linguistic analysis which uses N-branching trees where N > 2. Although such N-branching trees have been commonly used in liguistic analysis, unification operations to directly handle these analyses have not been proposed. Multiple-unification would unify feature structures propagated from each branch of trees simultaneously, and result in a considerable reduction in computational cost. This would benefit, particularly, parsing of Japanese where each case-marked NP can be subcategorized by VP at the top-level.

In our algorithm, multiple-unification is handled simply by assigning M markers for each tree or DAG identification where binary unification uses only markers 0 and 1. The algorithm itself should be changed by re-locating the failure detection stage to the end of the entire process, so that all merging is completed when failure detection is performed:

1. Shared Node Detection Stage
2. Merging Stage
3. Arc Merging Stage
4. Failure Detection Stage

Since unifiability of DAGs must be tested for all combinations, it is more efficient to merge first rather than to test unifiability N(N-1)/2 times before the merging stage.

8. Efficiency of the Algorithm

8.1. A Brief Complexity Analysis

The algorithm is efficient. Let's assume that we have DAGs with N nodes, depth D and width W. Shared node detection stage requires propagation of markers from roots to each value node. Since this can be done in parallel, computational cost is approximately D × (P + CSH) whereas P is a time required for propagation of a marker for one depth, and CSH is a cost for detecting wether two nodes has a same PUC. In the failure detection stage, back-propagation of markers for one travarsal backward is requried. The cost is P. The merging stage requires $2 \times D \times P + P$ at worst cases. The arc merging state costs $3 \times D \times P + P$ at worst cases. Thus, in total, $6 \times D \times P + 3 \times P + C + CSH \times D$ is the computational cost of the full unification in our algorithm with 2N-1 processors. Thus, in rough estimation, a complexity of the algorithm is of order of O(D). When the number of processors (M) is less than 2N-1, efficiency might degrade depending upon allocation of nodes onto processors. If we can allocate nodes in a same path to one processor, we only require

2: Back-Propagate(PT, PU(V,-S), 1, nil, nil, Mark(PT))
3: If there is a PU with PT and S, then Back-Propagate(Value, PU(PT,S), PU(R), nil, nil, ADD-Values)

Table 5: The failure detection stage of the fuzzy unification

W processors to maintain the efficiency close to the estimation above. This is because a marker-propagation in the same path is sequential. However, W processor condition may degrade its efficiency due to synchronization required for marker-propagation at arc merging and branching crossing processor boundary. The worst case of W processor condition is $O(\frac{D \times N}{W})$, but, of course, this can be easily avoided by designing memory allocation optimally. When unification failed, then the computational cost is $D \times P$ (cost for shared node detection) and P (cost for failure detection). Let S be a success rate of the unification (which is usually between 40% to 20%), expected computational costs will be: $S \times (D \times (6 \times P + CSH) + 3 \times P + C) + (1 - S) \times (D \times (P + CSH) + P)$

8.2. Experimental Results

We have implemented our algorithms on a simulator for a fine-grained parallel machine which assumes actual computation time for each instruction. To unify the DAGs shown in figure 4, the destructive graph unification took 1957 micro seconds (510 unification per second). The rate of performance degradation is about 330 micro seconds for each additional depth.

Table 6 shows numbers of each instruction executed, and computational cost in one example of the unification operation. Statistics clearly show that the shared node detection stage is the most computationally expensive. Particularly, the extensive numbers of address propoagation and bit check operation are two major causes of the computational cost. The estimated time for propagating an address for one traverse is set to 15 micro seconds, which can be reduced to 3 micro seconds on SNAP architecture, thus attining substantial speed up. Algorithms described in this paper has been implemented on the SNAP massively parallel computer as a part of the joint project between Carnegie Mellon University and University of Southern California.

9. Conclusion

This paper described unification algorithms using marker-passing. We only assumed passing of bit-markers, pointers to PUs, and values. Operations required for our unification algorithms are simple and easily implementable in massively parallel machines which use numbers of processing units with a relatively low-processing capability. Actually, operations and marker-passing schemes assumed in this paper are readily available in actual massively parallel machines such as SNAP [Moldovan et. al., 1989].

The algorithms are simple. It requires passing of bit-markers and addresses to PUs for conventional unifications. Despite its simplicity, our algorithms cover all non-disjunctive cases of unification of trees and most practical cases of unification of graphs. However, investigations should be conducted to identify a class of graphs which our algorithms can and cannot handle. Should a class of graphs which can be handled by our algorithms cover a class of graphs appearing in natural language processing, our algorithms can be a very powerful scheme of parallel unification processing. Typed-unification, originally proposed by [Ait-Kaci, 1984], can be naturally incorporated in our algorithms since our algorithms are based on marker-passing which is originally proposed for an intersection search. Conformity with lattice search is obvious.

The algorithms are efficient on massively parallel machines. Even in nondestructive graph unification, it requires only nine propagations and back-propagations and some checking instructions. For the graphs with depth D, unification should be done at $6 \times D \times P + 3 \times P + C + CSH \times D$ whereas P is a time required for propagation of a marker for one arc traverse, C is a total cost of condition checks, and CSH is a total cost for detecting whether two nodes has a same PUC. Thus, the complexity is of order of O(D). The processor requirement is linear to the size of graphs. This simple estimation indicates that our algorithm would be fast enough for practical applications.

Novel features such as multiple-unification and fuzzy unification adds new dimensions to our unification algorithms. Also, our unification algorithms are easily augmented for typed unification. In practical cases, needs for unification of more than two feature structures are commonly observed, yet this has not been proposed in the past. Use of multiple-unification reduces the amount of copying and thereby improves performance. Fuzzy unification would be a very useful concept for applications such as spoken language processing. Instead of rejecting at the detection of unification failure, the fuzzy unification adds a cost of violation in such cases, and allows processing of violated hypotheses to proceed. Where application domains inevitably involve ungrammatical inputs, the fuzzy-unification would be a powerful extension to the traditional unification approach.

Acknowledgement

The author would like to thank Hitoshi Iida, Hideto Tomabechi, Dan Moldovan, and members of the SNAP

Stage	Propagate Markers	Propagate Address	Bit Check	Address Check	Store Address	Time (micro-seconds)
Shared Node Detection	0	74	157	15	0	1778
Failure Detection	1	0	4	0	0	30
Merge	0	2	14	0	2	108
Internal Arc Merge	0	2	8	0	2	78
Total	1	78	183	15	4	1994

Table 6: Number of Instructions at each stage of unification

project for discussions, and Masaru Tomita and Jaime Carbonell for their supports.

References

[Ait-Kaci, 1984] Ait-Kaci, H., *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*, Ph.D. Thesis, University of Pennsylvania, 1984.

[Dwork et. al., 1984] Dwork, C., Kanellakis, P. and Mitchell, J., "On the Sequential Nature of Unification," *Journal of Logic Programming*, vol. 1, 1984.

[Hillis, 1985] Hillis, D., *The Connection Machine*, The MIT Press, Cambridge, 1985.

[Knight, 1989] Kevin, K., "Unification: A Multi-Disciplinary Survey," *ACM Computing Surveys, Vol. 21, Number 1*, 1989.

[Moldovan et. al., 1989] Moldovan, D., Lee, W., and Lin, C., *SNAP: A Marker-Propagation Architecture for Knowledge Processing*, University of Southern California Technical Report CENG 89-10, 1989.

[Nirenberg et. al., 1989] Nirenberg, S. (Ed.), *Knowledge-Based Machine Translation*, Center for Machine Translation Project Report, Carnegie Mellon University, 1989.

[Pollard and Sag, 1987] Pollard, C. and Sag, I., *An Information-based Syntax and Semantics*, Volume 1., Chicago University Press, 1987.

[Tomita and Knight, 1988] Tomita, M. and Kevin, K., "Pseudo-Unification and Full Unification," CMU-CMT-88-MEMO, 1988.

[Wroblewski, 1988] Wroblewski, D., "Nondestructive Graph Unification," in *Proceedings of AAAI-88*, 1988.

[Yasuura, 1984] Yasuura, H., "On Parallel Computational Complexity of Unification," in *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984.

UNIFICATION-BASED DEPENDENCY PARSING OF GOVERNOR-FINAL LANGUAGES

(Hyuk-Chul Kwon)

Dept. of Computer Science
College of Natural Science, Pusan National University
30 changjun-dong, Keumjung-ku, Pusan 609-735, Republic of Korea
Phone : 82-051-510-2218(office)
Phone : 82-051-556-6223(home)
E-mail : hckwon @ cosmos.kaist.ac.kr
Fax : 82-051-510-1792

(Aesun Yoon)

Dept. of French
College of Cultural Sciences, Pusan National University
30 changjun-dong, Keumjung-ku Pusan 609-735, Republic of Korea

ABSTRACT

This paper describes a unification-based dependency parsing method for governor-final languages. Our method can parse not only projective sentences but also non-projective sentences. The feature structures in the tradition of the unification-based formalism are used for writing dependency relations. We use a structure sharing and a local ambiguity packing to save storage.

This paper was supported in part by NON DIRECTED RESEARCH FUND, Korea Research Foundation, 1989

I. Introduction

The parsers of phrase structure grammars face troubles for parsing free word order languages in following respects.

First, they require a large size of grammatical rules for parsing free word order languages. *Second*, the free word order often results in discontinuous constituents(Covington, 1988). A phrase-structure tree of a sentence with discontinuous constituents would have crossing branches. This crossing branches can not be represented by conventional context free rules. *Third*, free word order languages feature very rich systems of morphological markings(Kwon, 1990). Word arrangements

and morphological markings are obviously contingent on relations between wordforms rather than on constituency(Mel'cuk, 1988).

One approach to parse free word order languages is the principle-based parsing(Berwick, 1987). The other approach is the dependency parsing(Mel'cuk, 1988).

This paper describes a unification-based dependency parsing method for governor-final(head-final) languages like Korean and Japanese. We develop the parsing method with special reference to Korean but the method can be adapted directly to Japanese parsing. Korean and Japanese are relatively free word order languages(Kwon, 1990). Although their word order is free except that dependents always precede their governor, word order variations lead to different emphasis on the topic and the focus. In contrast, their morpheme order is fixed at the level of words.

In Korean and Japanese, it is quite natural to drop any arguments including a subject and an object if they can be recovered through the context. Null subjects are also found in Italian and Spanish(Moon, 1989). Null arguments make it much harder to parse Korean and Japanese using phrase structure grammars. Because dependency grammars analyze syntactic structure as the relationships between ultimate syntactic units(i.e, morpheme, part of speech), dependency parsers can easily parse sentences with null arguments.

This paper follows the grammatical formalism of Mel'cuk(1988), but modifies it for computational efficiency and Korean specific characteristics. We try to parse not only projective sentences but also non-projective sentences. Feature structures in the tradition of unification-based grammars are used for writing dependency relations. But unification operation is modified for parsing non-projective sentences. A structure sharing and a local ambiguity packing is used to save storage.

II. Dependency Relations and Feature Structures

Mel'cuk differentiates three dependency relations : morphological dependency, syntactic dependency and semantic dependency(Mel'cuk, 1988).

The syntactic dependency is binary relations between wordforms, which are anti-symmetric, anti-reflexive and anti-transitive. The syntactic relations are represented by arcs : X -> Y: where X governs Y; X is called the governor of Y; and Y is called the dependent of X. The syntactic relations are best represented by a connected directed labeled graph.

Mel'cuk gives additional restrictions on the syntactic structure. First, a syntactic structure contains exactly one node(root) that does not depend on another node. Second, in a syntactic structure, no node may simultaneously depend on two or more other nodes. The syntactic structure becomes a rooted tree, specifically a D-tree by these two restrictions.

In Korean and Japanese, there are two different morphemes: free(content) morphemes and bound(function) morphemes. Bound morphemes include postpositions and verbal endings. A free morpheme can depend on another morpheme directly. But a bound morpheme can depend on another morpheme after it governs other morphemes. This means that the leaf nodes of the D-tree are always free morphemes.

We use feature structures in the tradition of unification-based grammars for writing dependency relations(Sells, 1985).

governor	relation	dependent
[cat : postposition]	case-marking	[cat : noun]
[cat : verb-stem]	actant	[cat : postposition attributive : - coordinative : -]
[cat : verbal-ending]	modal-marking	[cat : verb-stem]
[cat : noun]	attributive	[cat : postposition attributive : +]
[cat : noun]	coordinative	[cat : postposition coordinative : +]

< Table 1>

<Table 1> shows parts of the government pattern of Korean. As Korean and Japanese are governor-final languages, dependents always precede their governors. But there are no precedence relations between dependents in general.

[lex : "John"
cat : noun
animate : +]

[lex : "Susan"
cat : noun
animate : +]

[lex : "i"
cat : postposition
case : nominative
bound : +]

[lex : "ul"
cat : postposition
case : accusative
bound : +]

[lex : "po"
cat : verb-stem
{subcat=>subj,obj}
subj : [animate : +]]

[lex : " da"
cat : verbal-ending
modal : declarative
bound : +]

< Dictionary 1 >

<Dictionary 1 > is a sample Korean dictionary. The feature "bound" is used to differentiate between bound morphemes and free morphemes. When a bound morpheme governs another morpheme, the value of "bound" become "nil". As "bound" is not controlled by the unification operation, the change of the value of "bound" does not destroy the monotonicity of the unification. More explanation will be found in chapter III.

(1) John - i Susan - ul po - da
SM OM VS VE
(see) (DEC)

(2) Susan-ul John-i po-da

< SM : Subject Marker,OM :Object Marker, VS : Verb Stem, VE : Verbal Ending, DEC : DEClarative >

In (1) and (2), the subject marker("i") governs "John" and the object marker("ul") governs "Susan". "Po" governs both the nominative construction ("John-i") and the accusative construction ("Susan-ul"). Because of no dependency between "John-i" and "Susan-ul", there is no precedence relation between them. "da" governs "John-i Susan-ul po". As a result, both (1) and (2) are grammatical sentences and they have the same meaning as "John sees Susan".

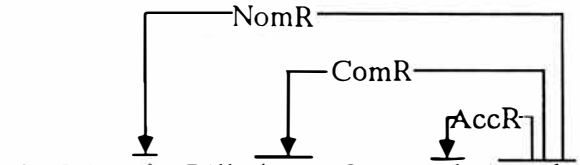
(3) John-i Bill-kwa Susan-ul po-da

Although "kwa" is a postposition, it can depend on a verb stem or a noun, but not both. When it depends on a verb stem, its meaning is "with". But its meaning is "and" if it depends on a noun. <Dictionary 2> shows the lexical information of "kwa".

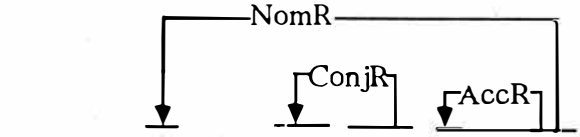
lex : "kwa" cat : postposition case : commitative bound : +	lex : "kwa" cat : postposition case : conjunctive coordinative : + bound : +	lex : "eui" cat : postposition case : possesive attributive : + bound : +

< Dictionary 2 >

From <Table 1> and <Dictionary 2>, we conclude that (3) has two different interpretations.



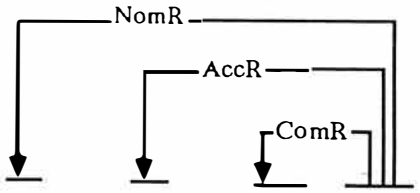
(3') John -i Bill -kwa Susan -ul po -da
(John sees Susan with Bill)



(3") John -i Bill -kwa Susan -ul po -da
(John sees Susan and Bill)

< NomR : Nominative Relation, AccR : Accusative R, ConjR :Conjunctive R, ComR :Comminative R>

But (4) has only one interpretation.

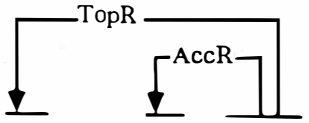


(4) John-i Susan-ul Bill-kwa po-da

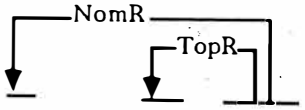
(John sees Susan with Bill)

<Table 1> and <Dictionary 2> also show that the possessive postposition "eui(of)" only depends on a noun.

The subcategorization of a verb gives additional constraints on the dependency relations. The subcategorization is used for a case assignment, the decision of null arguments and a filter on governing patterns. When a subject and an object are topicalized, the subject marker and the object marker are replaced to topic markers.



(5) John-un Susan-ul po-da
TM



(6) John-i Susan-un po-da
TM

< TM :Topic Marker, TopR : Topical R >

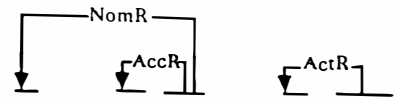
Postpositions do not provide the sufficient information for the case assignment of topicalized constructions in (5) and (6).

In (5), "po" governs the topicalized construction and the accusative construction. But verb stem "po" subcategorizes both a subject and an object. So, the noun of the topicalized construction is the subject of (5). (5) and (6) have the same meaning as (1) except that the subject and the object are topicalized respectively.

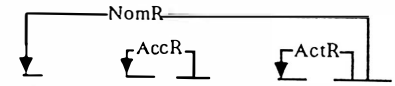
In Korean, the noun of a nominative construction is always the subject of a verb, and the noun of an accusative construction is the object of a verb, but not vice-versa. Therefore, we separate the case marking operation and the case assignment operation. The case of a topicalized construction is assigned when a verb stem is governed by a verbal ending.

(7) John-i Susan-ul po-ass - da - ko malha - da
VE VE VE VS VE
(past) (DEC) (COMP) (say) (DEC)
<COMP : COMPlementizer>

The decision of null arguments also requires the subcategorization. As the verb stem "malha" subcategorizes a subject and a complementizer("ko"), and "po" subcategorizes a subject and an object, two subjects are required in (7). But there is only one nominative construction. The nominative construction can be governed by "po" or "malha", but not both. As a result, we can conclude that one subject is dropped. (7) has two different interpretations as below.



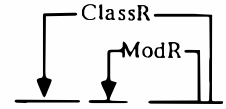
(7') John-i Susan-ul po - ass da-ko malha-da
(? says that John saw Susan)



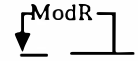
(7") John-i Susan-ul po-ass-da-ko malha - da
(John says that ? saw Susan)

< ? : null argument, ActR : Actant Relation >

Another constraints are required to parse the constructions with numerals of Korean and Japanese.



(8) i) sajen se kwon(three dictionaries)
NOUN DET NOUN
(dictionary)(three)Book.Form



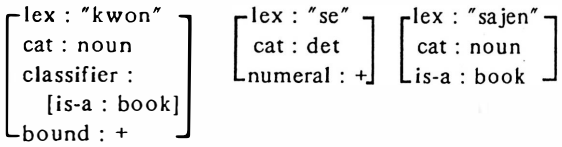
ii) se kwon (three book-like materials)
iii) *sajen kwon(not allowed)
iv) *se sajen kwon(not allowed)

<"kwon" : a unit for counting book-like materials, DET : determiner, ModR : Modificative Relation, ClassR : Classificative Relation>

"kwon" is a noun but a bound morpheme. We call it an incomplete noun. "kwon" can govern a numeral and a noun but there are restrictions in the governing order. "kwon" can govern a noun only after it governs a numeral, but the opposite is not true. This additional precedence restrictions can be formulated as <Table 2 > and <Dictionary 3>.

governor	relation	dependent
[cat :noun]	modificative	[cat:det]
[cat : noun, modifier [lex:det, numeral : +]]	classificative	[cat:noun]

< Table 2 >



< Dictionary 3 >

The second row of <Table 2> shows that a noun which is modified by a numeral (determiner) can govern a noun. The dictionary also shows that "kwon" is an incomplete noun and is a unit for counting books. There is a morphological dependency between "kwon" and "sajen". The above shows how our system deals with the morphological dependencies and additional precedence restrictions using feature structures.

III. Parsing Projective Sentences and Structure Sharing

Using dependencies for parsing natural languages, the projectivity is an extremely important property of the word order. A sentence is called projective if and only if the arcs of dependency links satisfy following restrictions(Mel'cuk, 1988).

(i) No arc crosses another arc

(ii) No arc covers the root of D-tree

Although most sentences of natural languages are projective, there exist several types of non-projective sentences. Non-projective sentences have discontinuous constituents. This chapter gives a parsing algorithm for projective sentences. The algorithm will be modified for non-projective sentences in the next chapter.

The algorithm scans a sentence from left-to-right for searching a governor. If a governor is found, it tries to make all the dependency links between the governor and the constructions whose head is the morpheme which immediately precedes the governor. The term *head* is used in the sense of top node of a construction as Mel'cuk(1988).

In a projective sentence, a governor can govern a wordform if and only if the governor governs directly or indirectly all the wordforms between them. Let $<m_1, m_2, \ldots, m_n>$ be an ordered list of morphemes. If m_i governs m_j and m_j governs m_k, then m_i indirectly governs m_k. The morpheme m_i can govern m_j if and only if all the morphemes between m_j and m_i are governed directly or indirectly by m_i where $j < i$. A head governs directly or indirectly all the other morphemes in a construction.

Our parsing strategy is as follows.

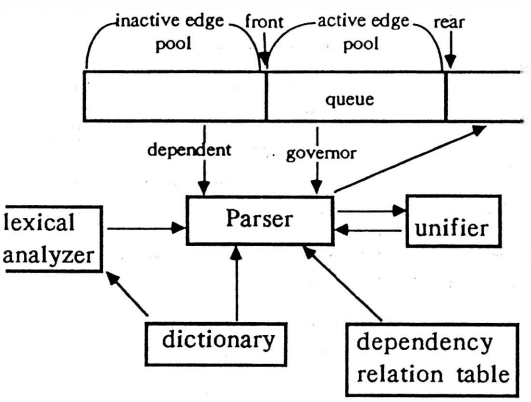
First) The parser gets a morpheme m_i from the lexical analyzer until an end-of-sentence marker is encounted.

Second) The parser searches constructions whose head is m_{i-1}. When there exist dependency relations between m_i and some of them, the parser generates new constructions and stores them in the queue.

Third) When some constructions exist in the queue, the parser gets one of them from the queue. Otherwise, goto *first)*. Let that construction contain all the morphemes from m_j to m_i where $j < i$ and m_i is its head. The parser searches construc-

tions whose head is m_{j-1}. When there exist dependency relations between m_i and some of them, the parser generates new constructions, stores them in the queue and repeats *third)*.

We implement the algorithm by chart. <Fig.1> shows the architecture of a Korean parser which runs at Apollo workstations.

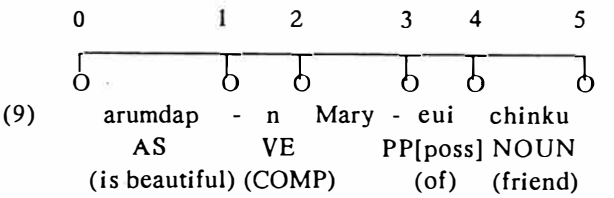


<Fig. 1>

The parser joins one dependent to one governor at a time. Each edge has a starting point and an ending point.

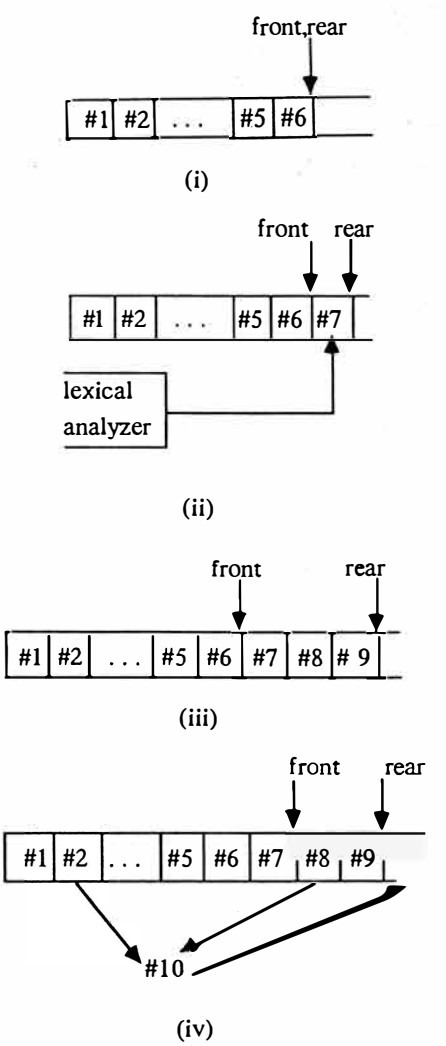
#	SP	EP	Construction	Remark
1	0	1	arumdap	
2	0	2	[arumdap,n]	
3	2	3	Mary	
4	0	3	[[arumdap,n],Mary]	beautiful Mary
5	0	4	[[[arumdap,n],Mary],eui]	of beautiful Mary
6	2	4	[Mary, eui]	of Mary
7	4	5	chinku	
8	2	5	[[Mary,eui],chinku]	friend of Mary
9	0	5	[[[[arumdap,n],Mary],eui],chinku]	friend of beautiful Mary
10	0	5	[[[arumdap,n],Mary,eui]],chinku]	beautiful friend of Mary

<Table 3>



<AS : Adjective Stem, VE : Verb Ending, PP[poss] : Possesive Postposition>

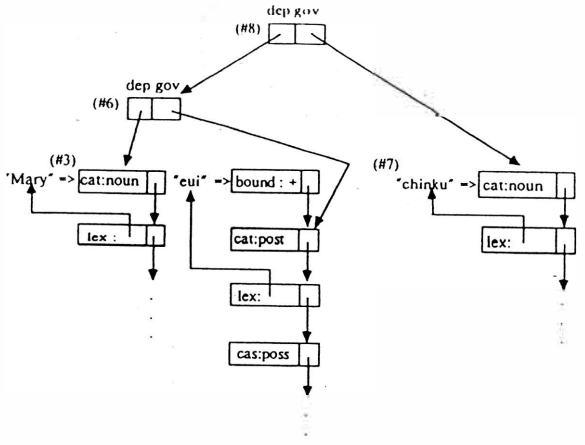
<Table 3> shows the content of the pool while (9) is parsed. (9) means "a/the friend of Mary who is beautiful" and has two different interpretations as (#9) and (#10). <Fig .2> shows the state of the pool when "chinku" is processed.



<Fig. 2> The state of the pool

(i) is the pool after processing "arumdap-n Mary-eui". As the inactive edge pool is empty, the parser gets "chinku" from the lexical analyzer as (ii). When the processing of (#7) is finished, the pool become (iii). (iv) shows the pool when (#10) is generated. As bound morphemes ("n","eui") can not depend on other morphemes by themselves, it is not necessary to store bound morphemes at the pool.

The storage for parsing grows exponentially as ambiguities are increased. We use a structure sharing(Tomita, 1986) and a local ambiguity packing(Shieber, 1986) to save storage. Although the order of the features is not important in the unification formalism, we always place the "bound" feature first.



<Fig. 3>

<Fig. 3> shows that (#8) shares the structures of (#6) and (#7). (#6) shares the structure of "eui" except for the "bound" feature. As the "bound" feature is excluded, the monotonicity of the unification is not destroyed.

We state that two or more subtrees represent a local ambiguity if they have the same starting point and the same ending point and if their top nodes are the same wordform. That is, (#9) and (#10) of the <Table 3> represent a local ambiguity. If a sentence has many local ambiguities, the total ambiguities would grow exponentially. To avoid this, we use a technique called local ambiguity packing which is suggested by Tomita(1986).

#	SP	EP	constructions
1	0	1	"arumdap"
2	0	2	[#1 | "n"]
3	2	3	"Mary"
4	0	3	[#2 | #3]
5	0	4	[#4 | "eui"]
6	2	4	[#3 | "eui"]
7	4	5	"chinku"
8	2	5	[#6 | #7]
9,10	0	5	[OR([#2,#6],#5)|#7]

[[#2,#6]|#7] = [#2|#8] = #10

<Table 4>

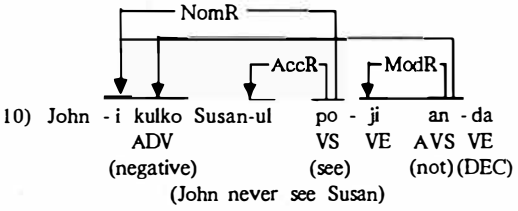
<Table 4> is the content of the pools after (9) is parsed with a structure sharing and a local ambiguity packing. #(9,10) in <Table 4> is the result of the local ambiguity packing of (#9) and (#10) in <Table 3>.

IV. Parsing Non-Projective Sentences

Non-projective sentences give serious difficulties in parsing natural languages.

But almost all languages have some sorts of non-projectivity(Mel'cuk, 1988).

There are two types of non-projectivity in Korean. The first one is related to the feature co-occurrence where the dependency links do not pass over the sentence boundary.



< *("kulko" ~ "an") = never, AVS:Auxiliary Verb Stem, ADV:ADVerb>

"kulko" is used only in negative sentences. In (10), "po" governs "John-i" and "Susan-ul", but the auxiliary verb stem "an" governs "kulko" and "po-ji". "kulko" can be placed anywhere before "an" at (10).

In a non-projective sentence, a governor can govern a wordform although the governor does not govern directly or indirectly some wordforms between them. This is one of the greatest obstacles for parsing non-projective sentences by our parsing method.

To overcome this problem, we introduce a new type feature called a co-occurrence feature. A co-occurrence feature-value is represented as ["fn" : c "v"], where "fn" is a feature name and "v" is a value. ["fn" : c "v"] means that its governor must have the feature-value ["fn" : "v"].

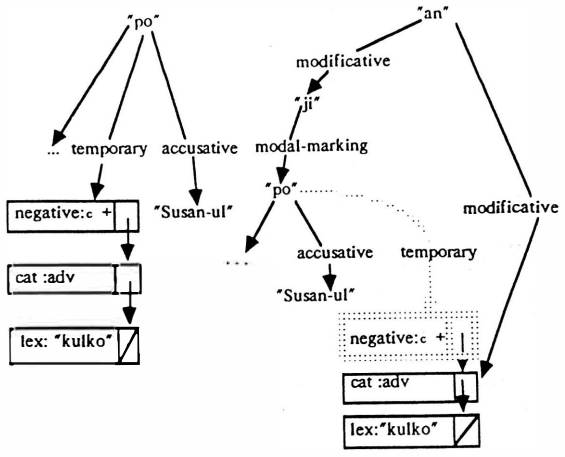
[negative : c +, cat : adverb, lex : "kulko"] [negative : +, cat : aux-verb-stem, lex : "an"]

<Dictionary 4>

governor	Relation	Dependent
[cat :verb-stem]	temporary	[negative : c +, cat :adverb]
[cat:aux-verb-stem, lex : "an"]	modificative	[negative : c +, cat :adverb]
[cat:aux-verb-stem]	modificative	[cat:verbal-ending]

<Table 5>

The first row of < Table 5 > shows that a verb stem temporarily governs an adverb which has [negative : c +]. When the verb stem depends on a construction which has [negative : +] and the dependency does not pass over the sentence boundary, the temporary dependency link is removed and a new dependency link between the adverb and the construction is connected. Two constructions are not unified when their dependency is temporary. We handle the co-occurrence feature similar to the "bound" feature.

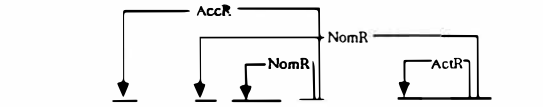


(i) (ii)

<Fig. 4>

When "an" governs "po-ji", a new link between "an" and "kulko" replaces the temporary link between "po" and "kulko". It is important that [negative: c +] is removed in (ii). If some co-occurrence features remain after the parsing, the sentence is incorrect.

The other type of non-projectivity occurs by non-local dependencies. Some constructions which are the dependents of an embedded verb can be placed at outer sentences in Korean. We can also find non-local dependencies in Finnish(Karttunen, 1986).



As stated above, "po" subcategorizes a subject and an object, and "malha" subcategorizes a subject and a complementizer. (11) has the cross arcs because "malha" governs "John-i" and "po" governs "Susan-ul".

Karttunen shows that this problem can be solved by functors with a floating type in Finish(Karttunen, 1986). The same framework also works in Korean. The framework can yield more than one results, but most of them are only acceptable at extraordinary situations. Therefore, our system strengthens the framework as a construction can be combined only with the nearest verb stem which can govern it when there is no projective governor of it.

V. Conclusion

We have shown a unification-based dependency parsing method for governor-final languages like Korean and Japanese. Feature structures in the tradition of unification-based grammars have been used for writing dependency relations. Our method can parse non-projective sentences as well as projecitve sentences.

We implement a Korean parser by the method presented in this paper using C language. The first version parser only used a structure sharing. But the current version uses a structure sharing and a local ambiguity packing. The local ambiguity packing saves about 35% of storage for parsing sample sentences.

More efficient structure sharing method and the dictionary structure are under study. We plan to use our method for parsing fixed word order languages.

Reference

[1] Berwick, R. C. 1987 Principle-Based Parsing, A.I. T.R. No. 972, MIT AI Lab.

[2] Covington, M. A. 1988 Parsing Variable Word Order Language with Unification-Based Dependency Grammar, ACMC Research Report 01-0022, University of Georgia.

[3] Hellwig, P. 1986 Dependency Unification Grammar, Proc. of Colling 86, pp.195-198.

[4] Karttunen, L. 1989 Radical Lexicalism, Alternative Conceptions of Phrase Structure, The University of Chicago Press., pp44-65.

[5] Kashket, M. B. 1987 A Government-Binding Based Parser for Warlpiri, TR 993, MIT AI Lab.

[6] Kwon, H., Yoon, A., Kim, Y. 1990 A Korean Analysis System Based on Unification and Chart, Proc. of Pacific Rim International Conference on Artificial Intelligence '90, pp251-256.

[7] Mel'cuk, I.A. 1988 Dependency Syntax : Theory and Practice, State of University of New York Press.

[8] Moon, G. 1989 Ph.D.Diss.,The Syntax of Null Arguments with Special Reference to Korean, The University of Texas at Austin.

[9] Sato, P. T. 1988 " A Common Parsing Scheme for Left- and Right-Branching Languages," J. of Computational Linguistics, Vol. 14, No.1, pp.20-30.

[10] Sells, P. 1985 Lectures on Contemporary Syntactic Theories, CSLI.

[11] Shieber,S.M et al. 1986 A Compilation of Papers on Unification-Based Grammar Formalisms : Part II, Report No. CSLI-86-48

[12] Tomita, M. 1986 Efficient Parsing for Natural Language, Kluwer Academic Pub.

February 15, 1991

Session B

Pearl: A Probabilistic Chart Parser*

David M. Magerman
CS Department
Stanford University
Stanford, CA 94305
magerman@cs.stanford.edu

Mitchell P. Marcus
CIS Department
University of Pennsylvania
Philadelphia, PA 19104
mitch@linc.cis.upenn.edu

Abstract

This paper describes a natural language parsing algorithm for unrestricted text which uses a probability-based scoring function to select the "best" parse of a sentence. The parser, Pearl, is a time-asynchronous bottom-up chart parser with Earley-type top-down prediction which pursues the highest-scoring theory in the chart, where the score of a theory represents the extent to which the context of the sentence predicts that interpretation. This parser differs from previous attempts at stochastic parsers in that it uses a richer form of conditional probabilities based on context to predict likelihood. Pearl also provides a framework for incorporating the results of previous work in part-of-speech assignment, unknown word models, and other probabilistic models of linguistic features into one parsing tool, interleaving these techniques instead of using the traditional pipeline architecture. In preliminary tests, Pearl has been successful at resolving part-of-speech and word (in speech processing) ambiguity, determining categories for unknown words, and selecting correct parses first using a very loosely fitting covering grammar.[1]

Introduction

All natural language grammars are ambiguous. Even tightly fitting natural language grammars are ambiguous in some ways. Loosely fitting grammars, which are necessary for handling the variability and complexity of unrestricted text and speech, are worse. The standard technique for dealing with this ambiguity, pruning grammars by hand, is painful, time-consuming, and usually arbitrary. The solution which many people have proposed is to use stochastic models to train statistical grammars automatically from a large corpus.

Attempts in applying statistical techniques to natural language parsing have exhibited varying degrees of success. These successful and unsuccessful attempts have suggested to us that:

- Stochastic techniques combined with traditional linguistic theories *can* (and indeed must) provide a solution to the natural language understanding problem.
- In order for stochastic techniques to be effective, they must be applied with restraint (poor estimates of context are worse than none[8]).
- Interactive, interleaved architectures are preferable to pipeline architectures in NLU systems, because they use more of the available information in the decision-making process.

We have constructed a stochastic parser, Pearl, which is based on these ideas.

The development of the Pearl parser is an effort to combine the statistical models developed recently into a single tool which incorporates all of these models into the decision-making component of a parser. While we have only attempted to incorporate a few simple statistical models into this parser, Pearl is structured in a way which allows any number of syntactic, semantic, and other knowledge sources to contribute to parsing decisions. The current implementation of Pearl uses Church's part-of-speech assignment trigram model, a simple probabilistic unknown word model, and a conditional probability model for grammar rules based on part-of-speech trigrams and parent rules.

By combining multiple knowledge sources and using a chart-parsing framework, Pearl attempts to handle a number of difficult problems. Pearl has the capability to parse word lattices, an ability which is useful in recognizing idioms in text processing, as well as in speech processing. The parser uses probabilistic training from a corpus to disambiguate between grammatically acceptable structures, such as determining prepo-

*This work was partially supported by DARPA grant No. N0014-85-K0018, ONR contract No. N00014-89-C-0171 by DARPA and AFOSR jointly under grant No. AFOSR-90-0066, and by ARO grant No. DAAL 03-89-C0031 PRI. Special thanks to Carl Weir and Lynette Hirschman at Unisys for their valued input, guidance and support.

[1]The grammar used for our experiments is the string grammar used in Unisys' PUNDIT natural language understanding system.

sitional phrase attachment and conjunction scope. Finally, Pearl maintains a well-formed substring table within its chart to allow for partial parse retrieval. Partial parses are useful both for error-message generation and for processing ungrammatical or incomplete sentences.

In preliminary tests, Pearl has shown promising results in handling part-of-speech assignment, prepositional phrase attachment, and unknown word categorization. Trained on a corpus of 1100 sentences from the Voyager direction-finding system[2] and using the string grammar from the PUNDIT Language Understanding System, Pearl correctly parsed 35 out of 40 or 88% of sentences selected from Voyager sentences not used in the training data. We will describe the details of this experiment later.

In this paper, we will first explain our contribution to the stochastic models which are used in Pearl: a context-free grammar with context-sensitive conditional probabilities. Then, we will describe the parser's architecture and the parsing algorithm. Finally, we will give the results of some experiments we performed using Pearl which explore its capabilities.

Using Statistics to Parse

Recent work involving context-free and context-sensitive probabilistic grammars provide little hope for the success of processing unrestricted text using probabilistic techniques. Works by Chitrao and Grishman[3] and by Sharman, Jelinek, and Mercer[14] exhibit accuracy rates lower than 50% using *supervised training*. Supervised training for probabilistic CFGs requires parsed corpora, which is very costly in time and man-power[2].

In our investigations, we have made two observations which attempt to explain the lack-luster performance of statistical parsing techniques:

- Simple probabilistic CFGs provide *general* information about how likely a construct is going to appear anywhere in a sample of a language. This average likelihood is often a poor estimate of probability.
- Parsing algorithms which accumulate probabilities of parse theories by simply multiplying them overpenalize infrequent constructs.

Pearl avoids the first pitfall by using a context-sensitive conditional probability CFG, where context of a theory is determined by the theories which predicted it and the part-of-speech sequences in the input sentence. To address the second issue, Pearl scores each theory by using the geometric mean of the contextual conditional probabilities of all of the theories which have contributed to that theory. This is equivalent to using the sum of the logs of these probabilities.

[2]Special thanks to Victor Zue at MIT for the use of the speech data from MIT's Voyager system.

CFG with context-sensitive conditional probabilities

In a very large parsed corpus of English text, one finds that the most frequently occurring noun phrase structure in the text is a noun phrase containing a determiner followed by a noun. Simple probabilistic CFGs dictate that, given this information, "determiner noun" should be the most likely interpretation of a noun phrase.

Now, consider only those noun phrases which occur as subjects of a sentence. In a given corpus, you might find that pronouns occur just as frequently as "determiner noun"s in the subject position. This type of information can easily be captured by conditional probabilities.

Finally, assume that the sentence begins with a pronoun followed by a verb. In this case, it is quite clear that, while you can probably concoct a sentence which fits this description and does not have a pronoun for a subject, the first theory which you should pursue is one which makes this hypothesis.

The context-sensitive conditional probabilities which Pearl uses take into account the immediate parent of a theory[3] and the part-of-speech trigram centered at the beginning of the theory.

For example, consider the sentence:

My first love was named Pearl.
(no subliminal propaganda intended)

A theory which tries to interpret "love" as a verb will be scored based on the part-of-speech trigram "adjective verb verb" and the parent theory, probably "S → NP VP." A theory which interprets "love" as a noun will be scored based on the trigram "adjective noun verb." Although lexical probabilities favor "love" as a verb, the conditional probabilities will heavily favor "love" as a noun in this context.[4]

Using the Geometric Mean of Theory Scores

According to probability theory, the likelihood of two *independent* events occurring at the same time is the product of their individual probabilities. Previous statistical parsing techniques apply this definition to the cooccurrence of two theories in a parse, and claim that the likelihood of the two theories being correct is the product of the probabilities of the two theories.

[3]The parent of a theory is defined as a theory with a CF rule which contains the left-hand side of the theory. For instance, if "S → NP VP" and "NP → det n" are two grammar rules, the first rule can be a parent of the second, since the left-hand side of the second "NP" occurs in the right-hand side of the first rule.

[4]In fact, the part-of-speech tagging model which is also used in Pearl will heavily favor "love" as a noun. We ignore this behavior to demonstrate the benefits of the trigram conditioning.

This application of probability theory ignores two vital observations about the domain of statistical parsing:

- Two constructs occurring in the same sentence are not necessarily independent (and frequently are not). If the independence assumption is violated, then the product of individual probabilities has no meaning with respect to the joint probability of two events.
- Since statistical parsing suffers from sparse data, probability estimates of low frequency events will usually be inaccurate estimates. Extreme underestimates of the likelihood of low frequency events will produce misleading joint probability estimates.

From these observations, we have determined that estimating joint probabilities of theories using individual probabilities is too difficult with the available data. We have found that the geometric mean of these probability estimates provides an accurate assessment of a theory's viability.

The Actual Theory Scoring Function

In a departure from standard practice, and perhaps against better judgment, we will include a precise description of the theory scoring function used by Pearl. This scoring function tries to solve some of the problems noted in previous attempts at probabilistic parsing[3][14]:

- Theory scores should not depend on the length of the string which the theory spans.
- Sparse data (zero-frequency events) and even zero-probability events do occur, and should not result in zero scoring theories.
- Theory scores should not discriminate against unlikely constructs when the context predicts them.

The raw score of a theory, θ is calculated by taking the product of the conditional probability of that theory's CFG rule given the context (where context is a part-of-speech trigram and a parent theory's rule) and the score of the trigram:

$$SC_{\text{raw}}(\theta) = \mathcal{P}(rule_\theta | (p_0p_1p_2), rule_{\text{parent}}) sc(p_0p_1p_2)$$

Here, the score of a trigram is the product of the mutual information of the part-of-speech trigram,[5] $p_0p_1p_2$, and the lexical probability of the word at the location of p_1 being assigned that part-of-speech p_1.[6] In the case of ambiguity (part-of-speech ambiguity or multiple parent theories), the maximum value of this product is used. The score of a partial theory or a complete theory is the geometric mean of the raw scores of all of the theories which are contained in that theory.

[5]The mutual information of a part-of-speech trigram, $p_0p_1p_2$, is defined to be $\frac{\mathcal{P}(p_0p_1p_2)}{\mathcal{P}(p_0xp_2)\mathcal{P}(p_1)}$, where x is any part-of-speech. See [4] for further explanation.

[6]The trigram scoring function actually used by the parser is somewhat more complicated than this.

Theory Length Independence This scoring function, although heuristic in derivation, provides a method for evaluating the value of a theory, regardless of its length. When a rule is first predicted (Earley-style), its score is just its raw score, which represents how much the context predicts it. However, when the parse process hypothesizes interpretations of the sentence which reinforce this theory, the geometric mean of all of the raw scores of the rule's subtree is used, representing the overall likelihood of the theory given the context of the sentence.

Low-frequency Events Although some statistical natural language applications employ backing-off estimation techniques[12][5] to handle low-frequency events, Pearl uses a very simple estimation technique, reluctantly attributed to Church[8]. This technique estimates the probability of an event by adding 0.5 to every frequency count.[7] Low-scoring theories *will* be predicted by the Earley-style parser. And, if no other hypothesis is suggested, these theories will be pursued. If a high scoring theory advances a theory with a very low raw score, the resulting theory's score will be the geometric mean of all of the raw scores of theories contained in that theory, and thus will be much higher than the low-scoring theory's score.

Example of Scoring Function As an example of how the conditional-probability-based scoring function handles ambiguity, consider the sentence

Fruit flies like a banana.

in the domain of insect studies. Lexical probabilities should indicate that the word "flies" is more likely to be a plural noun than an active verb. This information is incorporated in the trigram scores. However, when the interpretation

S → . NP VP

is proposed, two possible NPs will be parsed,

NP → noun (fruit)

and

NP → noun noun (fruit flies).

Since this sentence is syntactically ambiguous, if the first hypothesis is tested first, the parser will interpret this sentence incorrectly.

However, this will not happen in this domain. Since "fruit flies" is a common idiom in insect studies, the score of its trigram, noun noun verb, will be much greater than the score of the trigram, noun verb verb. Thus, not only will the lexical probability of the word "flies/verb" be lower than that of "flies/noun," but also the raw score of "NP → noun (fruit)" will be lower than

[7]We are not deliberately avoiding using all probability estimation techniques, only those backing-off techniques which use independence assumptions that frequently provide misleading information when applied to natural language.

that of "NP → noun noun (fruit flies)," because of the differential between the trigram scores.

So, "NP → noun noun" will be used first to advance the "S → . NP VP" rule. Further, even if the parser advances both NP hypotheses, the "S → NP . VP" rule using "NP → noun noun" will have a higher score than the "S → NP . VP" rule using "NP → noun."

Interleaved Architecture in Pearl

The interleaved architecture implemented in Pearl provides many advantages over the traditional pipeline architecture, but it also introduces certain risks. Decisions about word and part-of-speech ambiguity can be delayed until syntactic processing can disambiguate them. And, using the appropriate score combination functions, the scoring of ambiguous choices can direct the parser towards the most likely interpretation efficiently.

However, with these delayed decisions comes a vastly enlarged search space. The effectiveness of the parser depends on a majority of the theories having very low scores based on either unlikely syntactic structures or low scoring input (such as low scores from a speech recognizer or low lexical probability). In experiments we have performed, this has been the case.

The Parsing Algorithm

Pearl is a time-asynchronous bottom-up chart parser with Earley-type top-down prediction. The significant difference between Pearl and non-probabilistic bottom-up parsers is that instead of completely generating all grammatical interpretations of a word string, Pearl pursues the N highest-scoring incomplete theories in the chart at each pass. However, Pearl parses *without pruning.* Although it is only advancing the N highest-scoring incomplete theories, it retains the lower scoring theories in its agenda. If the higher scoring theories do not generate viable alternatives, the lower scoring theories may be used on subsequent passes.

The parsing algorithm begins with the input word lattice. An $n \times n$ chart is allocated, where n is the length of the longest word string in the lattice. Lexical rules for the input word lattice are inserted into the chart. Using Earley-type prediction, a sentence is predicted at the beginning of the sentence, and all of the theories which are predicted by that initial sentence are inserted into the chart. These incomplete theories are scored according to the context-sensitive conditional probabilities and the trigram part-of-speech model. The incomplete theories are tested in order by score, until N theories are advanced.[8] The resulting advanced theories are scored and predicted for, and the new incomplete predicted theories are scored and added to the chart. This process continues until an complete parse tree is determined, or until the parser decides, heuristically, that it should not continue. The heuristics we used for determining that no parse can be found for an input are based on the highest scoring incomplete theory in the chart, the number of passes the parser has made, and the size of the chart.

[8] We believe that N depends on the perplexity of the grammar used, but for the string grammar used for our experiments we used N=3. For the purposes of training, a higher N should be used in order to generate more parses.

Pearl's Capabilities

Besides using statistical methods to guide the parser through the parsing search space, Pearl also performs other functions which are crucial to robustly processing unrestricted natural language text and speech.

Handling Unknown Words Pearl uses a very simple probabilistic unknown word model to hypothesize categories for unknown words. When word which is unknown to the system's lexicon, the word is assumed to be any one of the open class categories. The lexical probability given a category is the probability of that category occurring in the training corpus.

Idiom Processing and Lattice Parsing Since the parsing search space can be simplified by recognizing idioms, Pearl allows the input string to include idioms that span more than one word in the sentence. This is accomplished by viewing the input sentence as a word lattice instead of a word string. Since idioms tend to be unambiguous with respect to part-of-speech, they are generally favored over processing the individual words that make up the idiom, since the scores of rules containing the words will tend to be less than 1, while a syntactically appropriate, unambiguous idiom will have a score of close to 1.

The ability to parse a sentence with multiple word hypotheses and word boundary hypotheses makes Pearl very useful in the domain of spoken language processing. By delaying decisions about word selection but maintaining scoring information from a speech recognizer, the parser can use grammatical information in word selection without slowing the speech recognition process. Because of Pearl's interleaved architecture, one could easily incorporate scoring information from a speech recognizer into the set of scoring functions used in the parser. Pearl could also provide feedback to the speech recognizer about the grammaticality of fragment hypotheses to guide the recognizer's search.

Partial Parses The main advantage of chart-based parsing over other parsing algorithms is that the parser can also recognize well-formed substrings within the sentence in the course of pursuing a complete parse. Pearl takes full advantage of this characteristic. Once Pearl is given the input sentence, it awaits instructions as to what type of parse should be attempted for this input. A standard parser automatically attempts to produce a sentence (S) spanning the entire input string. However, if this fails, the semantic interpreter might be able to derive some meaning from the sentence if given

non-overlapping noun, verb, and prepositional phrases. If a sentence fails to parse, requests for partial parses of the input string can be made by specifying a range which the parse tree should cover and the category (NP, VP, etc.).

The ability to produce partial parses allows the system to handle multiple sentence inputs. In both speech and text processing, it is difficult to know where the end of a sentence is. For instance, one cannot reliably determine when a speaker terminates a sentence in free speech. And in text processing, abbreviations and quoted expressions produce ambiguity about sentence termination. When this ambiguity exists, Pearl can be queried for partial parse trees for the given input, where the goal category is a sentence. Thus, if the word string is actually two complete sentences, the parser can return this information. However, if the word string is only one sentence, then a complete parse tree is returned at little extra cost.

Trainability One of the major advantages of the probabilistic parsers is trainability. The conditional probabilities used by Pearl are estimated by using frequencies from a large corpus of parsed sentences. The parsed sentences must be parsed using the grammar formalism which the Pearl will use.

Assuming the grammar is not recursive in an unconstrained way, the parser can be trained in an unsupervised mode. This is accomplished by running the parser without the scoring functions, and generating many parse trees for each sentence. Previous work[9] has demonstrated that the correct information from these parse trees will be reinforced, while the incorrect substructure will not. Multiple passes of re-training using frequency data from the previous pass should cause the frequency tables to converge to a stable state. This hypothesis has not yet been tested.[10]

An alternative to completely unsupervised training is to take a parsed corpus for any domain of the same language using the same grammar, and use the frequency data from that corpus as the initial training material for the new corpus. This approach should serve only to minimize the number of unsupervised passes required for the frequency data to converge.

Preliminary Evaluation

While we have not yet done extensive testing of all of the capabilities of Pearl, we performed some simple tests to determine if its performance is at least consistent with the premises upon which it is based. The test sentences used for this evaluation are ***not*** from the training data on which the parser was trained. Using Pearl's context-free grammar, these test sentences produced an average of 64 parses per sentence, with some sentences producing over 100 parses.

Unknown Word Part-of-speech Assignment

To determine how Pearl handles unknown words, we removed five words from the lexicon, *i, know, tee, describe,* and *station,* and tried to parse the 40 sample sentences using the simple unknown word model previously described.

In this test, the pronoun, *i,* was assigned the correct part-of-speech 9 of 10 times it occurred in the test sentences. The nouns, *tee* and *station,* were correctly tagged 4 of 5 times. And the verbs, *know* and *describe,* were correctly tagged 3 of 3 times.

pronoun	90%
noun	80%
verb	100%
overall	89%

Figure 1: Performance on Unknown Words in Test Sentences

While this accuracy is expected for unknown words in isolation, based on the accuracy of the part-of-speech tagging model, the performance is expected to degrade for sequences of unknown words.

Prepositional Phrase Attachment

Accurately determining prepositional phrase attachment in general is a difficult and well-documented problem. However, based on experience with several different domains, we have found prepositional phrase attachment to be a domain-specific phenomenon for which training can be very helpful. For instance, in the direction-finding domain, *from* and *to* prepositional phrases generally attach to the preceding verb and not to any noun phrase. This tendency is captured in the training process for Pearl and is used to guide the parser to the more likely attachment with respect to the domain. This does not mean that Pearl will get the correct parse when the less likely attachment is correct; in fact, Pearl will invariably get this case wrong. However, based on the premise that this is the less likely attachment, this will produce more correct analyses than incorrect. And, using a more sophisticated statistical model, this performance can easily be improved.

Pearl's performance on prepositional phrase attachment was very high (54/55 or 98.2% correct). The reason the accuracy rate was so high is that the direction-finding domain is very consistent in it's use of individual prepositions. The accuracy rate is not expected to be as high in other domains, although it certainly

[9] This is an unpublished result, reportedly due to Fujisaki at IBM Japan.

[10] In fact, for certain grammars, the frequency tables may not converge at all, or they may converge to zero, with the grammar generating no parses for the entire corpus. This is a worst-case scenario which we do not anticipate happening.

should be higher than 50% and we would expect it to be greater than 75 %, although we have not performed any rigorous tests on other domains to verify this.

Preposition	from	to	on	Overall
Accuracy Rate	92 %	100 %	100 %	98.2 %

Figure 2: Accuracy Rate for Prepositional Phrase Attachment, by Preposition

Overall Parsing Accuracy

The 40 test sentences were parsed by Pearl and the highest scoring parse for each sentence was compared to the correct parse produced by PUNDIT. Of these 40 sentences, Pearl produced parse trees for 38 of them, and 35 of these parse trees were equivalent to the correct parse produced by Pundit, for an overall accuracy rate of 88%.

Many of the test sentences were not difficult to parse for existing parsers, but most had *some* grammatical ambiguity which would produce multiple parses. In fact, on 2 of the 3 sentences which were incorrectly parsed, Pearl produced the correct parse as well, but the correct parse did not have the highest score.

Of the two sentences which did not parse, one used passive voice, which only occurred in one sentence in the training corpus. While the other sentence,

```
How can I get from cafe sushi to Cambridge
City Hospital by walking
```

did not produce a parse for the entire word string, it could be processed using Pearl's partial parsing capability. By accessing the chart produced by the failed parse attempt, the parser can find a parsed sentence containing the first eleven words, and a prepositional phrase containing the final two words. This information could be used to interpret the sentence properly.

Future Work

The Pearl parser takes advantage of domain-dependent information to select the most appropriate interpretation of an input. However, the statistical measure used to disambiguate these interpretations is sensitive to certain attributes of the grammatical formalism used, as well as to the part-of-speech categories used to label lexical entries. All of the experiments performed on Pearl thus far have been using one grammar, one part-of-speech tag set, and one domain (because of availability constraints). Future experiments are planned to evaluate Pearl's performance on different domains, as well as on a general corpus of English, and on different grammars, including a grammar derived from a manually parsed corpus.

Future work should also investigate Pearl's performance on speech data. By incorporating the speech recognizer's acoustic score into the parser's scoring function, one could investigate the parser's ability to select the appropriate word strings from an N-best list of a speech recognizer's output.

Conclusion

The probabilistic parser which we have described provides a platform for exploiting the useful information made available by statistical models in a manner which is consistent with existing grammar formalisms and parser designs. Pearl can be trained to use any context-free grammar, accompanied by the appropriate training material. And, the parsing algorithm is very similar to a standard bottom-up algorithm, with the exception of using theory scores to order the search.

More thorough testing is necessary to measure Pearl's performance in terms of parsing accuracy, part-of-speech assignment, unknown word categorization, idiom processing capabilities, and even word selection in speech processing. With the exception of word selection, preliminary tests show Pearl performs these tasks with a high degree of accuracy. But, in the absence of precise performance estimates, we still propose that the architecture of this parser is preferable to traditional pipeline architectures. Only by using an interleaved architecture can a speech recognizer efficiently make use of complex grammatical information to select from among hypothesized words.

References

[1] Ayuso, D., Bobrow, R., et. al. 1990. Towards Understanding Text with a Very Large Vocabulary. In Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley, Pennsylvania.

[2] Brill, E., Magerman, D., Marcus, M., and Santorini, B. 1990. Deducing Linguistic Structure from the Statistics of Large Corpora. In Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley, Pennsylvania.

[3] Chitrao, M. and Grishman, R. 1990. Statistical Parsing of Messages. In Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley, Pennsylvania.

[4] Church, K. 1988. A Stochastic Parts Program and Noun Phrase Parser for Unrestricted Text. In Proceedings of the Second Conference on Applied Natural Language Processing. Austin, Texas.

[5] Church, K. and Gale, W. 1990. Enhanced Good-Turing and Cat-Cal: Two New Methods for Estimating Probabilities of English Bigrams. *Computers, Speech and Language.*

[6] Church, K. and Hanks, P. 1989. Word Association Norms, Mutual Information, and Lexicography. In Proceedings of the 27th Annual Conference of the Association of Computational Linguistics.

[7] Fano, R. 1961. *Transmission of Information.* New York, New York: MIT Press.

[8] Gale, W. A. and Church, K. 1990. Poor Estimates of Context are Worse than None. In Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley, Pennsylvania.

[9] Hindle, D. 1988. Acquiring a Noun Classification from Predicate-Argument Structures. Bell Laboratories.

[10] Hindle, D. and Rooth, M. 1990. Structural Ambiguity and Lexical Relations. In Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley, Pennsylvania.

[11] Jelinek, F. 1985. Self-organizing Language Modeling for Speech Recognition. IBM Report.

[12] Katz, S. M. 1987. Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-35, No. 3.*

[13] Seneff, Stephanie 1989. TINA. In Proceedings of the August 1989 International Workshop in Parsing Technologies. Pittsburgh, Pennsylvania.

[14] Sharman, R. A., Jelinek, F., and Mercer, R. 1990. In Proceedings of the June 1990 DARPA Speech and Natural Language Workshop. Hidden Valley, Pennsylvania.

LOCAL SYNTACTIC CONSTRAINTS

Jacky Herz and Mori Rimon ([1]

The Computer Science Department
The Hebrew University of Jerusalem,
Giv'at Ram, Jerusalem 91904, ISRAEL

ABSTRACT

A method to reduce ambiguity at the level of word tagging, on the basis of local syntactic constraints, is described. Such "short context" constraints are easy to process and can remove most of the ambiguity at that level, which is otherwise a source of great difficulty for parsers and other applications in certain natural languages. The use of local constraints is also very effective for quick invalidation of a large set of ill-formed inputs. While in some approaches local constraints are defined manually or discovered by processing of large corpora, we extract them directly from a grammar (typically context free) of the given language. We focus on deterministic constraints, but later extend the method for a probabilistic language model.

1. Introduction: Local Constraints and their Use

Let S = W_1,..., W_N be a sentence of length N, {Wi} being the words composing the sentence. Ideally, a lexical-morphological analyzer can assign to each word W_i a unique tag t_i, expressing its grammatical characteristics (typically part of speech and features). The unique tag image t_1,..., t_N of S could then serve as input to NLP applications, including - but not limited to - parsing.

In reality, however, W_i may have more than one interpretation, hence t_i is not uniquely defined. Examples for ambiguity at this level in English are nouns (both in singular and in plural forms) which can be often interpreted at word-level as verbs; words ending with "ing" which are ambiguous between tentative readings as a progressive verb, a gerund and an adjective; etc. Hebrew, our main language of study, poses a much greater difficulty, because of the complexity of its morpho-syntax and the "terse" nature of the vowel-free writing system. In modern written Hebrew, nearly 60% of the words in running texts are ambiguous with respect to tagging, and the average number of possible readings of words in a running text is found to to be 2.4 (See [Francis 82] for data on English).[2] In addition, in many cases the morphological analysis of a Hebrew word yields a sequence of tags rather than a single tag, and different interpretations may be mapped to sequences of different lengths (similar phenomena may be found in other Semitic languages and in Romance languages where cliticization occurs). This is in fact a different order of the ambiguity issue. Consider as an example the written character string VRD (ורד), which can be interpreted in Hebrew as:

```
     [ Noun ]  ("vered" = a rose)
or:  [ Adj ]   ("varod" = rosy)
or:  [ Conj, Verb ] ("v-red" = and descend).
```

We will refer to a sequence of M tags ($M \geq N$) , which is a legal (per word) tag image corresponding to the sentence S = W_1,..., W_N , as a path. The number of potentially valid paths can

---

[1] The first author is also affiliated with the Open University.
The second author's main affiliation is the IBM Scientific Center, Haifa, Israel.
Please address e-mail correspondence to: rimon@hujics.BITNET rimon@haifasc3.IINUS1.IBM.COM

[2] The degree of ambiguity is obviously affected by the grain of the tagging system (the level of detail of the tag set).

be exponential in the length of the sentence if all words are ambiguous. A parser will reduce this number to the minimum feasible. But we are interested in quicker, even if not perfect methods to reduce the number of valid paths and word-level ambiguity.[3]

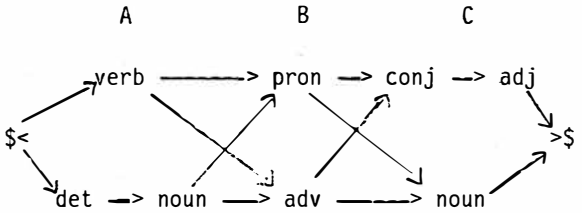
This paper describes a method to reduce tagging ambiguity, based on local syntactic constraints. A local constraint of length k on a given tag t is a rule disallowing a sequence of k tags from being in the Short Context of t. Intuitively, a Short Context with length k of a tag t in a given sentence S, denoted by SC(t,k), is its right or left tag environment. Before giving the formal definitions, let us mention that short context methods of one form or another are not new. They can be found in papers such as [Beale 88], [Choucka 85], [DeRose 88], [Katz 85], [Lozinskii 86], [Marcus 80], [Marshall 83] - to name a few. Our approach differs in various aspects, but mainly in the manner by which short context constrains are defined and identified. In the next chapter we will show how the constraints are retrieved directly from a grammar of the language, establishing a finite state mechanism which approximates the grammar.

To start with a more formal treatment of the short context notion, let us first add to the sentence S two special "words": "$<", denoting "Start" as the beginning of sentence marker, and ">$", denoting "End" at the end of sentence. These markers are also added to the tag image of the sentence.

We can now look at the resolution of ambiguity as a graph searching problem. As an example, suppose we have a sentence with three words, A B C, and assume that the initial tagging output of the lexical analyzer is the following (rather unlikely for English, but quite realistic for Hebrew):

```
for A:  [ verb ] or [ det, noun ]
for B:  [ pron ] or [ adv ]
for C:  [ conj, adj ] or [ noun ]
```

Then we can look at SG, the Sentence Graph, which is a directed graph where arcs represent all a-priori possible local paths:



Every path from "$<" to ">$" represents a possible interpretation of S as a stream of tags. Note that invalidating even a small number of arcs from SG reduces rapidly the number of possible paths.

As said above, we use local constraints to remove invalid arcs, and to finally arrive at the Reduced Sentence Graph.

Let T be the set of all possible tags - the tag set. The Right Short Context of length n of a tag t is defined by:

```
SCr (t,n)  for t in T and for n=0,1,2,3...

  { tz | z is in T* ,                          }
= {    |z| = n  or                             }
  {    |z| < n if ">$" is the last tag in z,   }
  {    and tz is a valid sequence of tags      }
```

The Left Short Context of length n of a tag t is denoted by SCl (t,n), and is defined in a symmetric way.

The definition of "validity" of tag sequences can vary. In our approach validity will be relative to a given formal grammar of the language, not to independent linguistic intuitions. This will be elaborated in the next chapter.

The Right (or Left) Positional Short Context i is the same as SCr (t,n) (or SCl (t,n)), but with the restriction that t may start only in position i in a sentence (or, in fact, in the tag image of a sentence). We denote the Right Positional

[3] Note that the two sub-goals of the tagging ambiguity problem - reducing the number of paths and reducing word-level possibilities - are not identical. One can easily construct sample sentences where each word is two-way ambiguous, hence the sentence has 2^N potential paths, of which only two are valid, while still keeping all word-level ambiguity.

Short Context of length n of a tag t in position i by PSCr (t,n,i); similarly, PSCl (t,n,i) denotes the Left Positional Short Context of length n of a tag t in position i.

The examples in this paper will refer to the Right Short Context (positional and non-positional) of length 1. This is done mainly for the sake of clarity, but empirically it seems that even the limited set of constraints which can be expressed in these terms is powerful enough to invalidate many arcs in the Sentence Graph, thus resolving a great deal of the tagging ambiguity. See also a comment to that effect in [Marshall 83].

In the ideal case, by removing arcs from the graph on the basis of local constraints, the reduced sentence graph will contain the one and only globally valid path from Start ("$ <") to End ("> $"). In such cases, all tag assignments are also uniquely determined. But there may be cases where several paths survive the short context tests, not only because there exist more than one legal syntactic analysis, but due to the fact that even illegal analyses at the sentence level may conform to local constraints. This means that some of the words may still have ambiguous tag assignments, and, if followed by parsing, the parser will have to rule out the (hopefully few) impossible combinations. There is another interesting case, where no path at all exists after reduction. This signifies an illegal input sentence; hence a quick and effective means to invalidate (at least part of the) illegal inputs.

The probabilistic model, which will be discussed in chapter 4, suggests a different scheme for reducing the sentence graph. Here arcs are not necessarily removed, but rather evaluated for relative plausibility. Only high probable overall path(s) through the graph will be selected.

2. Extracting Local Constraints from a Grammar

If a formal grammar G exists for the language L, then, by definition, it contains all the syntactic knowledge about L. As such, it also contains the knowledge about Short Contexts. However, most of this knowledge is not explicit; for example, boundary conditions (the adjacency of a final tag in a constituent phrase with the initial tag of the following phrase) are not explicitly stated in a phrase structure grammar; they have to be extracted to be used for preliminary screening of lexical and morphological ambiguities as described above.

In the following we will assume that an unrestricted context-free phrase structure grammar (CFG), G, exists for the given language L. Later we will discuss other grammars too. We will use the following notations:

T = The set of Terminal symbols (the tag-set)
$< = The sentence start terminal
> $ = The sentence end terminal
V = The set of Variables (non-terminals)
S = The root variable for derivations
P = Production rules of the form A --> α ,
where A is in V, α is in (V U T)*

For technical purposes, we will substitute every grammar rule of the form S --> α with an equivalent rule S --> $< α > $, thus adding the two special terminals mentioned above to T.

We will now revise the definitions of Short Context from chapter 1, relative to the given grammar G. The rules in G are the only source for determining the validity of tag sequences.

The Right Short Context of length n of a terminal t (tag) relative to the grammar G is defined by:

$$SC^{r}_{G}(t,n) \quad \text{for } t \text{ in } T \text{ and for } n=0,1,2,3\ldots$$

$$= \left\{ \begin{array}{l} tz \mid z \text{ is in } T^{*}, \\ \quad |z| = n \text{ or} \\ \quad |z| < n \text{ if ">\$" is the last tag in } z, \\ \quad \text{and there exists a derivation of the} \\ \quad \text{form: } S \Longrightarrow \alpha\ t\ z\ \beta \\ \quad \text{where } \alpha \text{ and } \beta \text{ are in } (V \cup T)^{*} \end{array} \right\}$$

The Left Short Context of length n of a terminal t (tag) relative to the grammar G is defined in a similar way, and is denoted by:

$$SC^{l}_{G}(t,n) \quad \text{for } t \text{ in } T \text{ and for } n=0,1,2,3\ldots$$

For short context with n = 1, it is useful (and natural) to define:

$$\text{next}(t) = \{ z \mid tz \text{ belongs to } SC^{r}_{G}(t,1) \}$$

The Right Positional Short Context of length n of a tag t in position i, relative to the grammar G, is defined by:

$$PSC^{r}_{G}(t,n,i) \quad \text{for } t \text{ in } T,\ n=0,1,2,3,\ldots,\ i>0$$

$$= \left\{ \begin{array}{l} tz \mid z \text{ is in } T^{*}, \\ \quad |z| = n \text{ or} \\ \quad |z| < n \text{ if ">\$" is the last tag in } z, \\ \quad \text{and there exists a derivation of the} \\ \quad \text{form: } S \Longrightarrow \alpha\ t\ z\ \beta \\ \quad \text{where } \alpha \text{ and } \beta \text{ are in } (V \cup T)^{*} \\ \quad \text{and } t \text{ is in the } i\text{-th position in a} \\ \quad \text{tag-image of a sentential form of } S \end{array} \right\}$$

The Left Positional Short Context is defined in a similar way and denoted by:

$$PSC^{l}_{G}(t,n,i)$$

The following is a procedure to compute the function next(t), from a CFG. Without loss of generality, one may assume that this CFG has no inaccessible symbols, has no useless symbols and is e-free, i.e. has no rules of the form V --> the empty string. [Aho 72] describes efficient algorithms to achieve this normal form.

We find the next(t) set by examining P, the rules of G:[4]

1. If there is a rule in P of the form:
   A --> α t x β and x is in T,
   then x is in next(t).

2. If there is a rule in P of the form:
   A --> α t B β and B is in V,
   then the set first(B) is
   a subset of next(t).

3. If there is a rule in P of the form:
   A --> α t
   then the set follow(A) is
   a subset of next(t).

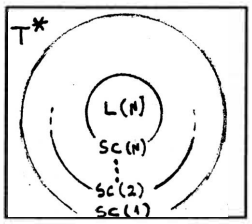
The computational complexity of the construction of the set next(t) depends on the complexity of computing the first and follow set. There are well known algorithms to find these sets from a given CFG. The complexity of follow(t) is exponential in the size of the look ahead window, which is the length of the context. This is another reason to limit the contexts to really short ones (although note that the extraction of constraints from the grammar is a one-time preprocessing phase, hence the performance issue is not critical).

To conclude this chapter, we borrow the concept of event dependency from probability theory, just to offer the following view on short context constraints. The events being concatenation of tags, the short context basically defines *independent* constraints, while in the full grammar the *dependent* constraints are expressed. This distinction is particularly apparent in SCr(t,1) or SCl(t,1), where "events" only apply to a pair of neighbors; as the context gets longer, the constraints become more dependent and closer to the full grammar. The metaphorical description above gets especially interesting when a statistical dimension is added to the model (see chapter 4). There, indeed, SC(1) considers independent probabilities of possible neighbors, where a full probabilistic grammar is supposed to look at the dependent events of tag concatenation along the full sentence.

It is therefore clear that the Short Context technique will license more sentences than a grammar would; or, from a dual point of view, it will invalidate only part of the impossible combinations of tag assignment. SCr(t,2) will have a closer fit coverage than SCr(t,1), and only in SCr(t,N) (where N is the finite length of a given sentence) the licensing power will be identical to the weak generative capacity of the full grammar

[4] The functions "first" and "follow" are used here much like in standard parsing techniques for both programming languages and natural languages; see [Aho 72] as a general reference.

(see illustration). However, SCr(t,N) has only the time complexity of a finite automaton (beware space complexity, though). The (theoretical and empirical) rate of convergence of the finite approximation is an interesting and important research topic. If indeed for a rather small number M, SCr(t, M) provides most of the licensing power of a given full grammar, then the performance promise of short context methods is consequential for a variety of applications (cf. [Church 80]). As mentioned before, it appears that even SCr(t,1) can drastically reduce the a-priori polynomial number of tag sequences, typically to a number linearly proportional to the length of the sentence.



3. An Example

Consider the following "toy grammar" for a small fragment of English (a variant of the basic sample grammar in [Tomita 86]).

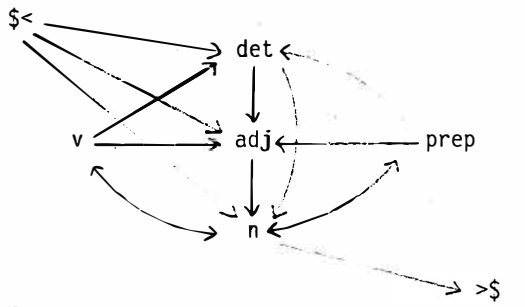
The tag set includes only: n (noun), v (verb), det (determiner), adj (adjective) and prep (preposition). The context free grammar G is:

```
S  --> $< NP VP >$
NP --> det n
NP --> n
NP --> adj n
NP --> det adj n
NP --> NP PP
PP --> prep NP
VP --> v NP
VP --> VP PP
```

G is a slightly modified version of a standard grammar, where the special symbols "$ <" (start) and "> $" (end) are added.

To extract the local constraints from this grammar, we first compute the function next(t) for every tag t in T, and from the result sets we obtain the graph below, showing valid moves in the short context of length 1 (validity is, of course, relative to the given toy grammar):

```
      r
The SC (t,1) Graph
      G
```



The table of valid neighbors is derived directly from the graph:

```
   r
SC (t,1) Table
  G
```

$<	det	adj	n
$<	n	det	adj
$<	adj	det	n
prep	det	n	v
prep	n	n	prep
prep	adj	n	>$
v	det		
v	n		
v	adj		

This table describes the closure of next(t) for all terminals in G.

Of special interest is the complement of the SC(t,1) table, relative to T^2 . Here, information about terminal pairs which can never appear in a legal sentence is represented. Such a table may be used by grammar developers to test a grammar, presenting small "checklist tests" which are easy to make.

From the SC(t,1) graph above we can now extract information about the Positional PSC(t,1,i) possibilities. This is done by tracing the way from "$ <" forward. The Positional Short Context tables are the following:

$$PSC_G^r(t,1,i)$$

Position:									
0 ---> 1		1 ---> 2		2 ---> 3		3 ---> 4			
$<	det	det	n	n	v	v	...		
$<	n	det	adj	n	prep	n	...		
$<	adj	n	prep	n	>$	prep	...		
		n	v	prep	det	det	...		
		n	>$	prep	n	adj	...		
		adj	n	prep	adj				
				v	det				
				v	n				
				v	adj				
				adj	n				

Note that from positions 3- > 4 on, the table gets identical to the general SC(t,l) table (the closure).

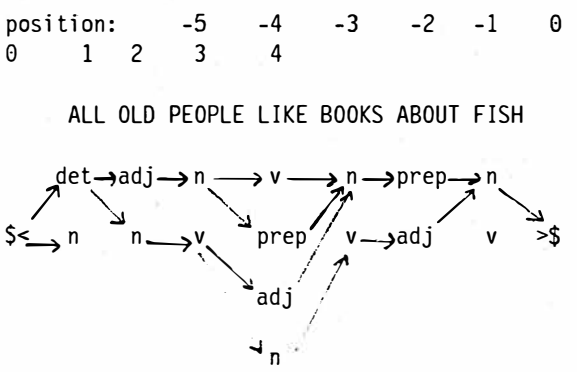
Another useful information one can obtain from the SC(t,l) graph is the inverse of the tables above - the Positional SC that may be allowed when going from the end of a sentence backwards. This is, in fact, the Positional Left Short Context. What has to be done to create the tables is to invert every arc in the SC(t,l) graph. Other than that, the procedure is the same. It is interesting to note that in our example the closure appears later when scanning the sentence backwards - from right to left.

A final technical comment before showing the operation on a sample sentence: When the short context of distinct occurrences of the same terminal is different, it is useful to distinguish between them using an index. This will add more information about the PSC when tracing the Sentence Graph.

Let us now consider the following sentence:

"All old people like books about fish."

The chart below shows the Reduced Sentence Graph - the original Sentence Graph from which invalid arcs (relative to the PSC tables) were removed.[5]



We are left with four valid paths through the sentence, out of 256 a-priori possible paths (256 = 2*2*2*4*2*2*2). Two paths represent legal syntactic interpretations (of which one is "the intended" meaning). The other two are locally valid but globally invalid (having either two verbs or no verb at all, in contrast to the grammar). SCr(t,2) would have invalidated one of the wrong two.

Note that in this particular example the method was quite effective in reducing sentence-wide interpretations (for applications like parsing), but it was not very good in individual word tagging disambiguation.

Finally, let us emphasize that, while it is not trivial to construct an interesting example in English to demonstrate all the above, in Hebrew, even relative to a grammar similar to the above, it is hard to find a written sentence without considerable ambiguity. Moreover, as mentioned earlier, Hebrew poses tagging ambiguity of a second order, where different-length tag sequences may be assigned to a single given word. But in graph terms, it only means that a certain sequence of tags can be represented as a sequence of linked vertices in SG, the sentence graph. Hence "second order ambiguity" does not present a problem to our method.

[5] The sentence is analyzed here relative to the limited tag set of the sample grammar. Depending on the tag set, the lexicon and the grammar, the level of ambiguity (and the results in this particular case) may be different.

4. Extensions

The method described above can be extended to be useful in a variety of situations other than those presented. In this chapter we briefly discuss several such extensions.

We already demonstrated how effective and efficient word tagging and path reduction can be used in a pre-parsing filter. We also mentioned applications (e.g. some types of proof-reading aids) which do not call for full parsing, but require "stand alone" tagging disambiguation and can benefit from fast recognition of many illegal inputs. On the other hand, for other applications, one may think of incorporation of short-context techniques directly into a parser. In such an environment, when the parser is about to test a hypothesis concerning the existence of a constituent, it will first check if local constraints do not rule out that hypothesis. The motivation is the same as that beyond different techniques combining top-down and bottom-up considerations. To render the method more effective, distinctions should be made between identical tags (terminals, categories) appearing in different constituents (phrase types). The process of extracting local constraints from the grammar can be changed to account for the required distinction (e.g. by indexing).

Another direction for extensions is to go beyond the model of straightforward context free grammars. The same process will hold as long as the short context can be easily computed from the grammar. The following are two such examples.

1. [Black 89] describes a process of transforming certain feature grammars into a finite state machine. The transition arcs in such a machine provide the full information required to construct our PSC tables.
2. Even when no efficient parser exists, L(SC) may still be easy to recognize. [Shamir 74] proved that testing membership in the family of the so-called context-free programmed languages is NP-complete; nevertheless, extracting local constraints from such grammars is easy. In fact, the recognition of L(SC) only depends on the existence of a formal grammar, not a parser.

We now turn to discuss a probabilistic language model, and see how short context considerations can be extended to account for probabilistic constraints.

In the probabilistic environment, adjacent tags are not only valid (1) or invalid (0), but are allowed in any given probability between 0 and 1. This model may be more realistic for NLP systems which process real-life texts, where some phenomena happen more frequently than others. The Short Context tables will therefore have to include weights.

We will first assume that a probabilistic context free grammar, such as described by [Fujisaki 89], [Wright 89] and others, exists for the given language. In a probabilistic CFG, rules are labeled by probability estimators. Typically, the sum of probabilities is 1 for all production rules sharing the same left hand side. The probability of a sentential form is computed from all estimators of the rules used in the process of derivation.

The probabilistic tables of the (Positional) Short Context can be extracted from such a grammar in various ways. The most natural (but not trivial!) method requires attachment and carrying over of probabilities through the procedure for calculating next(t), described in chapter 2. Another method to assign a probability to a tag pair [t1,t2], in a sentence image of n tags, could be based on evaluation of "dummy sentences" having t1 and t2 in positions i and i+1 respectively, and "wild card" entries elsewhere. But, since the probabilities attached to rules in the probabilistic CFG were most likely drawn from a corpus, it may make sense to calculate the short context information directly from the corpus, in parallel to the calculation of rule probabilities for the grammar. This is done by a simple counting of tag pairs appearing in successful analyses. To achieve a more natural normalization of statistical values, it may be better to define the weight of a tag pair in positions (i, i+1) in a sentence relative to all other possible tag pairs in the same positions. The method can be generalized for longer sequences of adjacent tags.

Similarly to the way a probabilistic CFG is constructed - by first defining the deterministic rules

and then attaching weights to rules - we can draw deterministic local constraints from a grammar and later assign relative frequency values to entries in the short context tables. Given a new sentence, one can first filter out all deterministically invalid arcs and only then evaluate paths in the reduced graph (where arcs are labeled with frequency estimators) for relative plausibility.

The resulting graph is similar to the notion of "span" in [Marshall 83] and [DeRose 88]. [DeRose 88] describes an efficient algorithm to find a plausible path in such graphs. The only difference is that our approach does not require unambiguous words to bound the scope of disambiguation - in our case the "$ <" and "> $" markers will define a scope of the full sentence.

Note that if no probabilistic grammar exists for the language, and even if there is no formal context free grammar available at all, but some operational parser is available, probabilistic constraints can still be drawn from a corpus. The process will involve analysis of sentences by the given parser, and counting of tag pairs (or longer tag sequences) present in successful analyses.[6] At the end of the corpus analysis, there will be a group of arcs for which the counter is still 0 (or below a given threshold). This may happen either because the arcs are indeed invalid - such arcs can be now removed completely from the tables (thus embedding, in fact, the deterministic method within the framework of the probabilistic one); or they may represent a marginal syntactic phenomenon in the text domain of the given corpus (here practical considerations will determine the decision whether to keep or to delete such arcs from the Short Context tables).

In this model it may be more convenient not to use probabilities, but rather to assign to each arc a rank, representing the complement of the counter relative to the largest one found. The larger the rank is, the less frequent (hence less plausible) is the corresponding arc.

A labeled sentence graph SG will now be created for input sentences, using these ranks. From this labeled graph, only the most probable path from start ($ <) to end (> $) is selected. For that, we suggest the algorithm by [Dijkstra 59], which efficiently finds the shortest weighted path between two vertices in a directed graph. In principle, one may want to identify more than the one most probable path, e.g. if the second best is also highly ranked. For that different (and more complex) algorithms are needed.

Note that the acquisition from a corpus described above brings the model very close to the corpus-based M-gram model, applied at the level of parts of speech; see [Katz 85], [Atwell 88], [Marshall 83], for accounts of related methods.

To conclude this chapter, we note that one may consider construction of deterministic grammars from corpora. Here the rules themselves will be defined based on data found in the text. Such grammars tend to be very large (cf. [Atwell 88]). Part of the reason is the grain of the tag set: such grammars might be inflated by the creation of "families" of very similar rules, not being able to recognize a generalization over similar tags. Another reason is in the distribution of rules (phrase structure) - only a small number of rules apply in a significant number of sample sentences, while most of the rules were derived from single examples. The performance efficiency of parsers (deterministic or probabilistic) based on such methods will greatly suffer from the large size of the grammar. But for the processing of local constraints, the size of the grammar is not terribly important. Once the preprocessing phase has been completed, the actual testing of constraints is not badly affected by the size of the constraints tables, thus making the local constraints approach effective in such an environment as well.

---

[6] It may not be absolutely required that only cases appearing in correct analyses are counted. Data resulting from wrong analyses may turn to be statistically insignificant, relative to real and frequent phenomena. cf. [Dagan 90].

5. Final Remarks

We have not attempted a rigorous discussion of the performance gains expected when applying tagging disambiguation in a pre-parsing filter and/or in the parsing process itself. The question is not easy. It strongly depends on the parsing technique, on one hand, and on the degree of ambiguity at the given language (as reflected in a given grammar), on the other hand. Naive bottom-up parsers, which assume a single combination of tags in each analysis pass against the grammar, can certainly benefit, by drastically reducing the exponential number of passes needed a-priori in cases of heavy ambiguity. Other more sophisticated parsing techniques (cf. [Kay 80], for example), can also save in computational complexity, by taking earlier decisions on inconsistent tag assignments and/or by requiring a smaller grammar. The detailed analysis here is not simple. But it seems that, although the constraints are drawn only from the grammar, and as such they are somehow expressed (explicitly or implicitly) and will take effect during parsing, the different order of computation and the restriction to finite-length considerations are sources for considerable time saving.

Another important question concerns properties of the grammar that help build an effective filter of tentative paths. The grain of the tag set is such a significant factor. A better refined tag set helps express more refined syntactic claims, but it also gives rise to a greater level of tagging ambiguity. It also requires a larger grammar (or longer lists of conditions on features, attached to phrase structure rules, which we here assume to be already reflected in the rules themselves), hence a larger set of local constraints. But these constraints will be much more specific and therefore more effective in resolving ambiguities. A rigorous analysis of this issue will help understand better what makes an effective disambiguator. An important point to make is that our method guarantees uniformity of the tag set used for the filter and for any parser acting upon the given grammar, thus making it useful in a variety of environments.

Acknowledgements

M. Bahr and E. Lozinskii gave us helpful comments and suggestions on earlier drafts of this paper. We gratefully acknowledge their contribution.

References

[**Aho 72**] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, 1972-3.

[**Atwell 88**] Eric S. Atwell and Clive Souter. Experiments with a Very Large Corpus-based Grammar. Proc. of *the 15th int'l conference of the ALLC*, Jerusalem, June 1988.

[**Beale 88**] Andrew David Beale. Lexicon and Grammar in Probabilistic Tagging of Written English. Proc. of *the 26th Annual Meeting of the ACL*, Buffalo NY, 1988.

[**Black 89**] Alan. W. Black. Finite State Machines from Feature Grammars. Proc. of *the 1st International Parsing Workshop*, Pittsburgh, June 1989.

[**Choueka 85**] Yaacov Choueka and Serge Lusignan. Disambiguation by Short Contexts. *Computers in the Humanities*, no. 3, Vol. 19, July 1985.

[**Church 80**] Kenneth W. Church. On Memory Limitations in Natural Language Processing. *MSc thesis, MIT*. 1980.

[**DeRose 88**] Steven J. DeRose. Grammatical Category Disambiguation by Statistical Optimization. *Computational Linguistics*, 14/1, Winter 1988.

[**Dagan 90**] Ido Dagan and Alon Itai. Processing Large Corpora for Reference Resolution. Proc. of *the 13th COLING conference*, Helsinki, 1990.

[**Dijkstra 59**] Edsgar W. Dijkstra. A Note on Two Problems in Connection With Graphs. *Numerische Mathematik*, 1, pp. 269-271.

[**Francis 82**] W. Nelson Francis and Henry Kucera. *Frequency Analysis of English Usage: Lexicon and Grammar*, Houghton-Mifflin, 1982.

[**Fujisaki 89**] T. Fujisaki, F. Jelinek, J. Cooke, E. Black, T. Nishimo. A Probabilistic Parsing Metohd for Sentence Disambiguation. Proc. of *the 1st International Parsing Workshop*, Pittsburgh, June 1989.

[**Greene 71**] Barbara Greene and Gerald Rubin. Automated Grammatical Tagging of English. *Technical Report*, Brown University,

1971.

[**Katz 85**] Slava Katz. Recursive M-gram Language Model via Smoothing of Turing Formula. *IBM Technical Disclosure Bulletin*, 1985.

[**Kay 80**] Martin Kay. Algorithm Schemata and Data Structures in Syntactic Processing. Report CSL-80-12, 1980. Reprinted in *Readings in Natural Language Processing*, Grosz, Sparck-Jones and Webber (eds.), Morgan Kaufman, 1986.

[**Lozinskii 86**] Eliezer L. Lozinskii and Sergei Nirenburg. Parsing in Parallel. *Comp. Languages*, UK, Vol 11, No. 1, pp 39-51, 1986.

[**Marcus 80**] Mitchell P. Marcus. *A Theory of Syntactic Recognition for Natural Language*, The MIT Press, 1980.

[**Marshall 83**] Ian Marshall. Choice of Grammatical Word-Class Without Global Syntactic Analysis: Tagging Words in the LOB Corpus. *Computers in the Humanities*, Vol. 17, pp. 139-150, 1983.

[**Shamir 74**] Eli Shamir and Catriel Beeri. Checking Stacks and Context Free Programmed Grammars Accept P-complete Languages. Proc. of *the 2nd Colloq. on Automata languages and programming*, Lecture Notes in Computer Science, Vol. 14, pp 27-33, 1974.

[**Tomita 86**] Masaru Tomita. *Efficient Parsing for Natural Language*, Kluwer Academic Pub., 1986.

[**Wright 89**] J. H. Wright and E. N. Wrigley. Probabilistic LR Parsing for Speech Recognition. Proc. of *the 1st International Parsing Workshop*, Pittsburgh, June 1989.

STOCHASTIC CONTEXT-FREE GRAMMARS FOR ISLAND-DRIVEN PROBABILISTIC PARSING

Anna Corazza(*), Renato De Mori(**), Roberto Gretter(*), Giorgio Satta(*)

(*) Istituto per la Ricerca Scientifica e Tecnologica,
38050 Povo di Trento (Italy)
(**) School of Computer Science, McGill University,
3480 University str, Montreal, Quebec, Canada, H3A2A7

ABSTRACT

In automatic speech recognition the use of language models improves performance. Stochastic language models fit rather well the uncertainty created by the acoustic pattern matching. These models are used to score *theories* corresponding to partial interpretations of sentences. Algorithms have been developed to compute probabilities for theories that grow in a strictly left-to-right fashion. In this paper we consider new relations to compute probabilities of partial interpretations of sentences. We introduce theories containing a gap corresponding to an uninterpreted signal segment. Algorithms can be easily obtained from these relations. Computational complexity of these algorithms is also derived.

1 INTRODUCTION

The aim of Automatic Speech Understanding (ASU) is to process an uttered sentence, determining an optimal word sequence along with its interpretation. The success of such a process depends on the formal system we use to model natural language. There is strong evidence that stochastic regular grammars (for example Markov Models) do not capture the large-scale structure of natural language. In very recent years, there has been a growing interest toward more powerful stochastic rewriting systems, like stochastic context-free grammars (SCFG's; see among the others [Wright and Wrigley 89], [Lari and Young 90], [Jelinek *et al.* 90] and [Jelinek and Lafferty 90]). Stochastic grammars fit naturally the uncertainty created by the (pattern matching) acoustic search process; moreover SCFG's give syntactic prediction capabilities that are stronger than the Markov Models. Further motivations for this approach are reported in [Lari and Young 90] and [Jelinek *et al.* 90].

In ASU we are interested in generating partial interpretations of a spoken sentence called *theories*. We score them in terms of their likelihood $L(A, th) = O(\Pr(A \mid th)\Pr(th))$,[1] where $\Pr(A \mid th)$ is the probability that theory th derives the acoustic signal segment A and $\Pr(th)$ is the probability of the obtained theory. The most popular parsers used in Automatic Speech Recognition (ASR) generate and expand theories starting from the left and then proceeding rightward. In this case, the best theories already obtained can drive the analysis of the right portion of the input, restricting the class of possible next preterminals in order to maximize the probabilities of the new extended theories. For ASU, especially for dialogue systems, it may be useful to consider parsers that are "island-driven". These parsers focus on *islands*, that is words of particular semantic relevance which have been previously hypothesized with high acoustic evidence. Then they proceed outward, working in both directions. Island-driven approaches have been proposed and defended in [Woods 81] and [Giachin and Rullent 89]; in [Stock *et al.* 89] the predictive power of bidirectional parsing is also discussed. None of the parsers proposed in these works uses a stochastic grammar.

In this paper we consider the problem of scoring partial theories in the island-driven approach. An important quantity is $\Pr(th)$, i.e. the probability that a SCFG generates sequences of words

[1] We write $f(x) = O(g(x))$ whenever there exist constants c, $\overline{x} > 0$ such that $f(x) > c\, g(x)$ for every $x > \overline{x}$.

(islands) separated by *gaps*. The gaps are portions of the acoustic signal that are still uninterpreted in the context of *th*. We develop a theoretical framework to compute $\Pr(th)$ in the case *th* contains islands and gaps.

2 NOTATION AND DEFINITIONS

In this section definitions related to Stochastic Context Free Grammars (SCFGs) are introduced, along with the notation that will be used throughout this paper.

An SCFG is defined as a quadruple $G_s = (\mathrm{N}, \Sigma, P, S)$, where N is a finite set of *nonterminal* symbols, Σ is a finite set of *terminal* symbols disjoint from N, P is a finite set of *productions* of the form $H \rightarrow \alpha$, $H \in \mathrm{N}$, $\alpha \in (\Sigma \cup \mathrm{N})^*$, and $S \in \mathrm{N}$ is a special symbol called *start symbol*. Each production is associated with a probability, indicated with $\Pr(H \rightarrow \alpha)$. The grammar G_s is *proper* if the following relation holds:

$$\sum_{\alpha \in (\Sigma \cup N)^*} \Pr(H \rightarrow \alpha) = 1, \quad H \in \mathrm{N}. \tag{1}$$

An SCFG G_s is in *Chomsky Normal Form* (CNF) if all productions in G_s are in one of the following forms:

$$H \rightarrow FG \quad H \rightarrow w, \quad H, F, G \in \mathrm{N}, w \in \Sigma. \tag{2}$$

For reasons discussed in [Jelinek *et al.* 90] it is useful to have the SCFG in CNF; in the following we will always refer to SCFGs in CNF.

The derivation of a string by the grammar G_s is usually represented as a parse (or derivation) tree, whose nodes indicate the productions employed in the derivation itself. It is also possible to associate with each derivation tree the probability that it was generated by the grammar G_s. This probability is the product of the probabilities of all the rules employed in the derivation.

Given a string $z \in \Sigma^*$, the notation $H < z >$, $H \in \mathrm{N}$, indicates the set of all trees with root H generated by G_s and spanning z. Therefore $\Pr(H < z >)$ is the sum of the probabilities of these subtrees, i.e. the probability that the string z has been generated by G_s starting from symbol H. We assume that the grammar G_s is *consistent* [Gonzales and Thomason 78]. This means that the following condition holds:[2]

$$\sum_{z \in \Sigma^*} \Pr(S < z >) = 1. \tag{3}$$

From this hypothesis it follows that a similar condition holds for all nonterminals.

A possible application of an island driven parser to a task of ASU is the following. On the basis of a previously obtained theory (partial interpretation) $u = w_i \dots w_{i+p}$ and of some non-syntactic knowledge, predictions can be made for words not necessarily adjacent to u. This introduces a gap within the theory that represents a not yet recognized part of the input sentence. Then further syntactical and acoustical analyses will try to fill in the gap. The gap will be then filled by further syntactical and acoustical analysis. Therefore we will deal with theories that can be represented as follows:

$$\begin{aligned} th: \quad & w_i \dots w_{i+p} x_1 \dots x_m w_j \dots w_{j+q} y_1 \dots y_k \dots \\ \text{or} \quad & u x^{(m)} v y^{(*)} \end{aligned} \tag{4}$$

where $w_i \dots w_{i+p} = u$ and $w_j \dots w_{j+q} = v$ indicate strings of already recognized terminals ($i, j > 0, p, q \geq 0, j > i+p$) while $x_1 \dots x_m = x^{(m)}$, $m \geq 0$ and $y_1 \dots y_k \dots = y^{(*)}$ stand for gaps with specified length m ($x^{(m)}$) or (finite) unspecified length ($x^{(*)}$). We will also indicate a gap with x meaning that either $x = x^{(m)}$ or $x = x^{(*)}$. In our notation, i and j are position indices, p and q are shift indices, m indicates a (known) gap length and k, h are used as running indices. Finally, Σ^* represents the set of all strings of finite length over Σ, while $\Sigma^m, m \geq 0$ is the set of all strings in Σ^* of length m.

[2]The normalization property expressed in (1) above guarantees that the probabilities of all (finite and infinite) derivations sum to one, but the language generated by the grammar only corresponds to the subset of the finite derivations, whose probability can be less than one.

We studied both the cases in which gap x has specified or unspecified length (see [Corazza *et al.* 90]). In practical cases, it is possible to estimate from the acoustic signal the probability distribution of the number of words filling the gap. Since this makes more significant the case in which the gap length is specified, in this work we will focus our attention on theories of the form $x = ux^{(m)}vy^{(*)}$.

3 PARTIAL DERIVATION TREE PROBABILITIES

For the calculation of the probability $\Pr(S < uxvy^{(*)} >)$, called *prefix-string-with-gap probability*, we use some quantities already introduced by other authors, like the *inside probability* $\Pr(H < u >)$ [Baker 79], [Lari and Young 90], [Jelinek *et al.* 90] or the *prefix-string probability* $\Pr(H < ux >)$ [Jelinek and Lafferty 90]. In [Jelinek and Lafferty 90] an algorithm is proposed for the computation of the latter probability in the case of unspecified gap length ($\Pr(H < ux^{(*)} >)$). We sketch here a similar algorithm for the cases in which the gap length equals m.

3.1 Prefix-string and Suffix-string probabilities

In the case of a known length gap $x^{(m)}$, a prefix-string probability $\Pr(H < ux^{(m)} >)$ can be computed on the basis of the following relation. Since G_s is in Chomsky Normal Form, if $|ux^{(m)}| > 1$ then H must directly derive two nonterminals G_1 and G_2. According to the way the string $ux^{(m)}$ can be divided into two parts spanned by G_1 and G_2 respectively, one can distinguish two different situations: in the first one, G_1 spans just a proper prefix of u and G_2 spans the remaining part of u and the gap; in the second one, G_1 entirely spans u plus a possible prefix of the gap. Based on these cases, the following relation can be established:

$$\Pr(H < ux^{(m)} >) = \sum_{G_1G_2} \Pr(H \to G_1G_2)[$$
$$\sum_{k=0}^{p-1} \Pr(G_1 < w_i \dots w_{i+k} >) \times$$
$$\times Pr(G_2 < w_{i+k+1} \dots w_{i+p}x^{(m)} >) +$$
$$+ \sum_{k=0}^{m-1} \Pr(G_1 < ux_1^{(k)} >)\Pr(G_2 < x_2^{(m-k)} >)]. \quad (5)$$

Note that gap $x^{(m)}$ has been split into two shorter gaps $x_1^{(k)}$ and $x_2^{(m-k)}$. By a recursive application of (5), prefix-string probabilities can be computed using both the following initial condition:[3]

$$\Pr(H < w_i x^{(0)} >) = \Pr(H \to w_i) \quad (6)$$

and the *gap probabilities* $\Pr(H < x^{(m)} >)$, which are the sum of the probabilities of all trees with root H and yield of length m. Gap probabilities can be recursively computed as follows:

$$\Pr(H < x^{(m)} >) = \sum_{G_1,G_2 \in N} \Pr(H \to G_1G_2) \times$$
$$\times \sum_{j=1}^{m-1} \Pr(G_1 < x^{(j)} >)\Pr(G_2 < x^{(m-j)} >), \; m > 1. \quad (7)$$

$$\Pr(H < x^{(1)} >) = \sum_{w \in \Sigma} \Pr(H \to w); \quad (8)$$

In a similar way we can define $\Pr(< xv >)$ as the *suffix-string probability*; its computation can be easily obtained from expressions that are symmetrical with respect to the ones employed for the prefix-string probability. Details are not pursued here.

We introduce now two probabilities that will be useful in calculating the prefix-string-with-gap probability: the *gap-in-string probability* $\Pr(H < uxv >)$ and the *island probability* $\Pr(H < xvy^{(*)} >)$.

3.2 Gap-in-string probabilities

For the gap-in-string probability computation we can distinguish three independent and mutually

[3] By convention, $x^{(0)}$ is the null string ε, i.e. the string whose length is zero.

exclusive cases, according to the position of the boundary between the two parts of string uxv spanned by the two children G_1 and G_2 of H. The first word of the string spanned by G_2 can belong to the initial string $u = w_i \dots w_{i+p}$, to the gap x or to the final string $v = w_j \dots w_{j+q}$.

In the case of known length gap one gets:

$$\begin{aligned}
&\Pr(H < w_i \dots w_{i+p} x^{(m)} w_j \dots w_{j+q} >) = \\
&= \sum_{G_1 G_2} \Pr(H \to G_1 G_2)[\\
&\quad \sum_{k=0}^{p-1} \Pr(G_1 < w_i \dots w_{i+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{i+k+1} \dots w_{i+p} x^{(m)} v >) + \\
&+ \quad \sum_{k=0}^{m} \Pr(G_1 < u x_1^{(k)} >) \Pr(G_2 < x_2^{(m-k)} v >) + \\
&+ \quad \sum_{k=0}^{q-1} \Pr(G_1 < u x^{(m)} w_j \dots w_{j+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{j+k+1} \dots w_{j+q} >)]. \qquad (9)
\end{aligned}$$

The inner summations in (9) contain products of already defined probabilities, along with terms that can be computed recursively with the following initial condition ($p = q = 0$):

$$\begin{aligned}
&\Pr(H < w_i x^{(m)} w_j >) = \sum_{G_1, G_2} \Pr(H \to G_1 G_2) \times \\
&\times \quad \sum_{k=0}^{m} \Pr(G_1 < w_i x_1^{(k)} >) \Pr(G_2 < x_2^{(m-k)} w_j >) \qquad (10)
\end{aligned}$$

3.3 Island probabilities

As for the gap-in-string case, the island probability computation involves three cases, depending on the position of the first word of the string spanned by G_2 with respect to the island $v = w_j \dots w_{j+q}$. The three sets of strings generated in the three cases above are probabilistically independent, but not disjoint in the case of unspecified length gap. Due to this fact, in such a case one must also consider the probability products, then obtaining a quadratic system of equations. On the other hand, the following relation is obtained for the case of m-length gap:

$$\begin{aligned}
&\Pr(H < x^{(m)} w_j \dots w_{j+q} y^{(*)} >) = \sum_{G_1, G_2} \Pr(H \to G_1 G_2)[\\
&\quad \sum_{k=1}^{m} \Pr(G_1 < x_1^{(k)} >) \times \\
&\qquad \times \Pr(G_2 < x_2^{(m-k)} w_j \dots w_{j+q} y^{(*)} >) + \\
&+ \quad \sum_{k=0}^{q-1} \Pr(G_1 < x^{(m)} w_j \dots w_{j+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{j+k+1} \dots w_{j+q} y^{(*)} >) + \\
&+ \quad \Pr(G_1 < x^{(m)} w_j \dots w_{j+q} y_1^{(*)} >) \times \\
&\qquad \times \Pr(G_2 < y_2^{(*)} >)] \qquad (11)
\end{aligned}$$

where the term $\Pr(G_2 < y_2^{(*)} >)$ equals 1. Using the definition of $Q_L(H \Rightarrow G_1\ G_2)$ given in [Jelinek and Lafferty 90] one can solve the recursion in (11) in the same way the recursive equation for the prefix-string probability is solved there, obtaining:

$$\begin{aligned}
&\Pr(H < x^{(m)} w_j \dots w_{j+q} y^{(*)} >) = \\
&= \sum_{G_1, G_2} Q_L(H \Rightarrow G_1 G_2) C_{xvy}(G_1, G_2) \qquad (12)
\end{aligned}$$

in which:

$$\begin{aligned}
&C_{xvy}(G_1, G_2) = \\
&= \sum_{k=1}^{m} \Pr(G_1 < x_1^{(k)} >) \times \\
&\qquad \times \Pr(G_2 < x_2^{(m-k)} w_j \dots w_{j+q} y^{(*)} >) + \\
&+ \quad \sum_{k=0}^{q-1} \Pr(G_1 < x^{(m)} w_j \dots w_{j+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{j+k+1} \dots w_{j+q} y^{(*)} >). \qquad (13)
\end{aligned}$$

The term $C_{xvy}(G_1, G_2)$ contains a summation of products between gap probabilities and island probabilities over a left gap shorter than x, along with a summation of products between suffix-string probabilities (with known length gap) and prefix-string probabilities (with unspecified length gap). Equation (13) can be solved recursively, with the initial condition ($x^{(0)} = \varepsilon$):

$$C_{vy}(G_1, G_2) = \sum_{k=0}^{q-1} \Pr(G_1 < w_j \dots w_{j+k} >) \times$$
$$\times \Pr(G_2 < w_{j+k+1} \dots w_{j+q} y^{(*)} >). \quad (14)$$

3.4 Prefix-string-with-gap probabilities

An expression for the prefix-string-with-gap probability $\Pr(\ H\ <\ ux^{(m)}vy^{(*)}\ >\ )$ can now be obtained directly from the four cases where the boundary between the two children of H belongs to u, to the gap x, to the island v or to the final gap y:

$$\begin{aligned}
&\Pr(H < w_i \dots w_{i+p} x^{(m)} w_j \dots w_{j+q} y^{(*)} >) = \\
&= \sum_{G_1, G_2} \Pr(H \rightarrow G_1 G_2)[\\
&\quad \sum_{k=0}^{p-1} \Pr(G_1 < w_i \dots w_{i+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{i+k+1} \dots w_{i+p} x^{(m)} v y^{(*)} >) + \\
&+ \quad \sum_{k=0}^{m} \Pr(G_1 < u x_1^{(k)} >) \Pr(G_2 < x_2^{(m-k)} v y^{(*)} >) + \\
&+ \quad \sum_{k=0}^{q-1} \Pr(G_1 < u x^{(m)} w_j \dots w_{j+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{j+k+1} \dots w_{j+q} y^{(*)} >) + \\
&+ \quad \Pr(G_1 < u x^{(m)} v y_1^{(*)} >) \Pr(G_2 < y_2^{(*)} >)]. \quad (15)
\end{aligned}$$

Solving the recursion in (15) in the same way as for (11), one obtains:

$$\Pr(H < w_i \dots w_{i+p} x^{(m)} w_j \dots w_{j+q} y^{(*)} >) = \sum_{G_1, G_2} Q_L(H \Rightarrow G_1 G_2) D_{uxvy}(G_1, G_2) \quad (16)$$

where:

$$\begin{aligned}
&D_{uxvy}(G_1, G_2) = \\
&\quad \sum_{k=0}^{p-1} \Pr(G_1 < w_i \dots w_{i+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{i+k+1} \dots w_{i+p} x^{(m)} v y^{(*)} >) + \\
&+ \quad \sum_{k=0}^{m} \Pr(G_1 < u x_1^{(k)} >) \Pr(G_2 < x_2^{(m-k)} v y^{(*)} >) + \\
&+ \quad \sum_{k=0}^{q-1} \Pr(G_1 < u x^{(m)} w_j \dots w_{j+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{j+k+1} \dots w_{j+q} y^{(*)} >). \quad (17)
\end{aligned}$$

As for previous computations in this section, equation (17) consists of summations over products of already defined probabilities along with a recursive term $\Pr(G_2 < w_{i+k+1} \dots w_{i+p} x^{(m)} v y^{(*)} >)$ which can be computed starting with the initial condition $(p = 0)$:

$$\begin{aligned}
&D_{w_i xvy}(G_1, G_2) = \\
&= \quad \sum_{k=0}^{m} \Pr(G_1 < w_i x_1^{(k)} >) \Pr(G_2 < x_2^{(m-k)} v y^{(*)} >) + \\
&+ \quad \sum_{k=0}^{q-1} \Pr(G_1 < w_i x^{(m)} w_j \dots w_{j+k} >) \times \\
&\qquad \times \Pr(G_2 < w_{j+k+1} \dots w_{j+q} y^{(*)} >). \quad (18)
\end{aligned}$$

4 COMPLEXITY EVALUATION

Based on the relation presented in the last section, algorithms for the computation of the probabilities defined there can be developed strightforwardly. In the present section we discuss the computational complexity for the cases of major interest (details about the derivation of the complexity expressions are simple but tedious, and therefore will not be reported here). The assumed model of computation is the *Random Access Machine*, taken under the *uniform cost criterion* (see [Aho *et al.* 74]). We are mainly concerned here with worst-case time complexity results.

We will indicate with $|P|$ the size of set P, i.e. the number of productions in G_s. All the probabilities defined in Section 3 depend upon the grammar G_s, strings u and v and the lengths of gaps x and y. Table 1 summarizes worst-case time complexity for sets of these probabilities.

computed set	*time complexity*
island probabilities	
1. $\{\Pr(H < x^{(m)} w_j \dots w_{j+q} y^{(*)} >) \mid H \in \mathrm{N}\}$	$O(\lvert P\rvert \max\{q^3, m^2 q\})$
prefix-string-with-gap probabilities	
2. $\{\Pr(H < w_i \dots w_{i+p} x^{(m)} w_j \dots w_{j+q} y^{(*)} >) \mid H \in \mathrm{N}\}$	$O(\lvert P\rvert \max\{p^3, q^3, pm^2, qm^2\})$
one word extension for island probabilities	
3. $\{\Pr(H < x^{(m)} w_j \dots w_{j+q} a y^{(*)} >) \mid H \in \mathrm{N}\}$	$O(\lvert P\rvert \max\{q^2, m^2\})$
4. $\{\Pr(H < x^{(m-1)} a w_j \dots w_{j+q} y^{(*)} >) \mid H \in \mathrm{N}\}$	$O(\lvert P\rvert \max\{m^2 q, mq^2\})$
one word extension for prefix-string-with-gap probabilities	
5. $\{\Pr(H < w_i \dots w_{i+p} x^{(m)} w_j \dots w_{j+q} a y^{(*)} >) \mid H \in \mathrm{N}\}$	$O(\lvert P\rvert \max\{p^2, q^2, m^2, (m+q)p\})$
6. $\{\Pr(H < w_i \dots w_{i+p} x^{(m-1)} a w_j \dots w_{j+q} y^{(*)} >) \mid H \in \mathrm{N}\}$	$O(\lvert P\rvert \max\{p^2 q, pq^2, p^2 m, pm^2\})$

Table 1: *Worst-case time complexity for the computation of the probabilities of some sets of theories. Symbol $a \in \Sigma$ indicates a one word extension of a theory whose probability had already been computed.*

Both island and prefix-string-with-gap probabilities require cubic time computations (rows 1 and 2). Rows 3 to 6 account for cases in which one have to compute the probability of a theory that has been obtained from a previously analyzed theory by means of a single word extension. In these cases, using a dynamic technique, one can dispense from the computation of elements already involved in the calculation of the previous theory. One word extension on the side of the unknown length gap $y^{(*)}$ costs quadratic time both in the case of island and prefix-string-with-gap probabilities. The one word extension on the side of the known length gap $x^{(m)}$ costs cubic time. This asymmetry can be justified observing from (15) that the addition of a single word between a string and a bounded gap forces the reanalysis of a quadratic number of new subterms. Note that this is also true for well known dynamic methods for CFG recognition (e.g. the CYK algorithm [Younger 67]): one word change in the *middle part* of a string implies a cubic-time whole recomputation in the worst-case. In fact there is an interesting parallelism between those methods, the Inside algorithm and the methods discussed here (see [Corazza *et al.* 90] for a discussion).

5 DISCUSSION

A framework has been developed to score partial sentence interpretations in ASU systems. General motivations for modeling natural language by SCFG's can be found in [Jelinek *et al.* 90], while the importance of scoring measures that are compatible with island-driven strategies has been already pointed out in [Woods 81]. In the present section we discuss major advantages of the studied approach and possible applications of the derived framework.

We are mainly interested in sentence interpretation systems. Even if semantical and pragmatical predictive models are not defined, we can rely on high-level heuristic information sources. This knowledge can be used to predict words on the base of previous partial interpretations. Predictions may be words not adjacent to the stimulating segments. These words can be recovered using word-spotting techniques.[4] Thus, the only way to employ the available heuristic information is to parse sentences in a discontinuous way. This means that the parser has first to find an island and then to fill the gap between the stimulating segment and the island itself. This technique produces partial analyses that are interleaved by gaps and that can be scored using our method.

[4] Word-spotting techniques allow one to find occurences of one (or more) given word in a speech signal. In these systems there is a trade off between "false alarms" and "missing words" that can be controlled by a threshold obtained from training speech.

The framework introduced in this paper can also be used to predict words adjacent to an already recognized string and to compute the probability that the first (last) word x_1 (x_m) of a gap is a certain symbol $a \in \Sigma$. This new word will extend the current theory. Words adjacent to an existing theory can be hypothesized by selecting the word(s) which maximize the prefix-string-with-gap probability of the theory augmented with it. Instead of computing these probabilities for all the elements in the dictionary, it is possible to restrict this expensive process to the preterminal symbols (as in [Jelinek and Lafferty 90]). The approach discussed so far should be compared with standard lattice parsing techniques, where no restriction is imposed by the parser on the word search space (see, for example [Chow and Roukos 89] and the discussion in [Moore *et al.* 89]).

Our framework accounts for bidirectional expansion of partial analyses; this improves the predictive capabilities of the system. In fact, bidirectional strategies can be used in restricting the syntactic search space for gaps surrounded by two partial analyses. This point has been discussed in [Stock *et al.* 89] for cases of one word length gaps. We propose a generalization to m-length gaps and to cases where partial analyses do not represent only complete parse trees but also partial derivation trees.

As a final remark, notice that the proposed framework requests the SCFG to be in Chomsky normal form. Although every SCFG G_s can be cast in CNF, such a process may result in quadratic size expansion of G_s, where the size of G_s is roughly proportional to the sum of the length of all productions in G_s. The proposed framework can be easily generalized to other kinds of bilinear forms with linear expansion in the size of G_s (for example the *canonical two form* [Harrison 78]). This consideration deserves particular attention because in natural language applications the size of the grammar is considerably larger than the input sentence length.

References

[Aho *et al.* 74] A.V.Aho, J.E.Hopcroft and J.D.Ullman: "The Design Analysis of Computer Algorithms", Addison-Wesley Publishing Company, Reading, MA, 1974.

[Baker 79] J.K.Baker: "Trainable Grammars for Speech Recognition", Proceedings of the Spring Conference of the Acoustical Society of America, 1979.

[Chow and Roukos 89] Y.L.Chow and S.Roukos: "Speech Understanding Using a Unification Grammar", Proceedings of the IEEE International Conference on Acoustic, Speech and Signal Processing, 1989, Glasgow, Scotland.

[Corazza *et al.* 90] A.Corazza, R.DeMori, R.Gretter and G.Satta: "Computation of Probabilities for an Island-Driven Parser" Technical Report SOCS 90-19, Mc Gill University, MONTREAL, Quebec, H3A 2A7 CANADA. Also as Technical Report TR9009-01, IRST, Trento, Italy, 1990.

[Giachin and Rullent 89] E.P.Giachin and C.Rullent: "A Parallel Parser for Spoken Natural Language", Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 1989, Detroit, Michigan USA, pp.1537-1542.

[Gonzales and Thomason 78] R.C.Gonzales and M.G.Thomason: "Syntactic Pattern Recognition", Addison-Wesley Publishing Company, Reading, MA, 1978.

[Harrison 78] M.A.Harrison: "Introduction to Formal Language Theory", Addison-Wesley Publishing Company, Reading, MA, 1978.

[Jelinek *et al.* 90] F.Jelinek J.D.Lafferty and R.L.Mercer: "Basic Method of Probabilistic Context Free Grammars", Internal Report, T.J.Watson Research Center, Yorktown Heights, NY 10598, 85 pages.

[Jelinek and Lafferty 90] F.Jelinek and J.D.Lafferty: "Computation of the Probability of Initial Substring Generation by Stochastic Context Free Grammars", Internal Report, Continuous Speech Recognition Group, IBM Research, T.J.Watson Research Center, Yorktown Heights, NY 10598, 10 pages.

[Lari and Young 90] K.Lari and S.J.Young: "The Estimation of Stochastic Context-Free Grammars using the Inside-Outside Algorithm" Computer Speech and Language, vol.4, n.1, 1990, pp.35-56

[Moore *et al.* 89] R.M.Moore F.Pereira and H.Murveit: "Integrating Speech and Natural Language Processing", Proceedings of the Speech and Natural Language Workshop, 1989, Philadelphia, Pennsylvania, pp.243-247.

[Stock *et al.* 89] O.Stock R.Falcone and P.Insinnamo: "Bidirectional Chart: A Potential Technique for Parsing Spoken Natural Language Sentences" Computer Speech and Language, vol.3, n.3, 1989, pp.219-237.

[Woods 81] W.A.Woods: "Optimal Search Strategies for Speech Understanding Control" Artificial Intelligence, vol.18, n.3, 1981, pp.295-326.

[Wright and Wrigley 89] J.H.Wright and E.N.Wrigley: "Probabilistic LR Parsing for Speech Recognition" International Workshop on Parsing Technologies, Pittsburg, PA, pp.105-114.

[Younger 67] D.H.Younger: "Recognition and Parsing of Context-Free Languages in Time n^3" Information and Control, vol. 10, 1967, pp.189-208.

February 15, 1991

Session C

Substring Parsing for Arbitrary Context-Free Grammars

Jan Rekers
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: rekers@cwi.nl

Wilco Koorn
Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

Abstract

A substring recognizer for a language L determines whether a string s is a substring of a sentence in L, i.e., *substring-recognize*(s) succeeds if and only if $\exists v, w\colon vsw \in L$. The algorithm for substring recognition presented here accepts general context-free grammars and uses the same parse tables as the parsing algorithm from which it was derived. Substring recognition is useful for *noncorrecting* syntax error recovery and for incremental parsing. By extending the substring *recognizer* with the ability to generate trees for the possible contextual completions of the substring, we obtain a substring *parser*, which can be used in a syntax-directed editor to complete fragments of sentences.

1 Introduction

A recognizer for a language L determines whether a sentence s belongs to L. A substring recognizer performs a more complicated job, as it determines whether s can be *part* of a sentence of L.

A recently developed substring recognition algorithm [4] uses an ordinary LR parsing algorithm with special parse tables. For ordinary parsing, this parsing algorithm is limited to LR(1) grammars, but the more complicated nature of substring recognition limits it to bounded-context grammars (see Section 3).

In Section 4 we describe a substring recognition algorithm that does not suffer from this drawback. It accepts general context-free grammars and uses the same parse tables as the ordinary parser. Our algorithm is based on the pseudo-parallel parsing algorithm of Tomita [17], which runs a dynamically varying number of LR parsers in parallel and accepts general context-free grammars. In Section 5 we extend the substring *recognizer* into a substring *parser* that generates trees for the possible completions of the substring.

Partial support received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II - GIPE II) and from the Netherlands Organization for Scientific Research – NWO, project *Incremental Program Generators*.

2 Applications

2.1 Syntax error recovery

In its simplest form, a parser stops at the first syntax error found. If it has to find as many errors in the input as possible, it can try to correct the error in order to continue parsing. Spurious errors are easily introduced, however, if the parser makes false assumptions about the kind of error encountered.

Substring parsing can be used to implement *non*correcting syntax error recovery. If an ordinary parser detects a syntax error on some symbol, the substring parser can be started on the next symbol to discover additional syntax errors. Using this method, it is not necessary to let the parser make any assumption about how to correct the error, or to let it skip input until a trusted symbol is found.

Richter defines noncorrecting syntax error recovery with the aid of substring parsing and interval analysis in a formal framework [15]. He proves that his technique does not generate spurious errors, but is not explicit about its implementation.

He notes, however, that there are difficulties in keeping the substring parser deterministic due to a limitation on the class of grammars accepted. Our technique could be useful here, as it implements the required substring analysis for general context-free grammars.

2.2 Completion tool

In Section 5 we will show how the substring recognizer can be extended so that it generates parse trees for the possible completions of a substring. As the total number of possible completions will often be infinite, only generic completions are generated. A syntax-directed editor could use these to complete fragments of sentences in accordance with the grammar used, or to guess the continuation of what the user is typing.

2.3 Incremental parsing

Another application for substring parsing is in incremental parsing. Incremental parsing can be performed by attaching parser states to tokens [3, 1, 18]. After a modification has been made, the parser is restarted in a saved state, at a point in the text just before the modification. Parsing stops when the parser reaches a token after the modification in an old configuration (if ever). These methods are very good as to minimizing the amount of recomputation after a modification, but require a huge amount of memory for storing the states of the parser (parse stacks with partial parse trees as elements).

Ghezzi and Mandrioli present an alternative technique for incremental parsing. [7, 8] If the string $x\tilde{x}z\tilde{y}y$ is modified to $x\tilde{x}\hat{z}\tilde{y}y$, where $\tilde{x}$ and $\tilde{y}$ have length k, with k the look-ahead used by the parser, then the parse trees previously generated for x and y are still valid after the modification. All subtrees previously generated for x and y can thus be abbreviated by their top nonterminals, which minimizes the length of the string to be reparsed. This technique is both time and space efficient, but is not applicable to general context-free parsing as it requires a fixed look-ahead. In our particular case, we need incremental parsing in a syntax-directed editor that uses the Tomita parser. By running a varying number of LR-parsers in parallel, the Tomita parser adjusts its look-ahead dynamically to the amount needed, and is thus not limited to an a priori known k.

Incremental parsing can also be achieved in another manner: after a modification has been made in the text, find the substring s' belonging to the smallest subtree that contains the modification in the stored parse tree. If the type of this subtree is T and s' can be parsed as a tree of type T, replace the old subtree by the new one. If s' fails to parse, it may be the case that the modification introduced a syntax error, or that the subtree has been chosen too small. These two cases must be distinguished, as the incremental parser proceeds in a different way in each case. A substring parser can provide a hint as to which of the two possibilities is actually the case. If the substring parser fails on s', the modification will be syntactically incorrect in any context, and an error message can be given. If the substring parser succeeds, a larger subtree is chosen and parsing is retried. This can be more time consuming than remembering parser states, but the amount of memory needed is far less. We consider using this scheme in the syntax-directed editor GSE [11], but it has to be investigated further as a lot of work is still performed twice.

3 Related work

Cormack [4] describes a substring parse technique for Floyd's class of bounded context or BC(1,1) grammars [6], and implements the substring parser Richter mentions [15]. A grammar is BC(1,1) if for every rule $A ::= \alpha$, if some sentential form contains $a\alpha b$ where α is derived from A then α is derived from A in *all* sentential forms containing $a\alpha b$. This class is smaller than LR(1). The solution of Cormack consists in using an ordinary LR automaton, but a special parse table constructor. The sets of items generated do not only contain items of the form $A ::= \alpha \bullet \beta$ but also "suffix items" of the form $A ::= \cdots \bullet \beta$. These suffix items denote partial handles whose origins occur before the beginning of the input. The generated parse tables are deterministic, provided that the grammar is BC(1,1). This substring parser is used for noncorrecting error recovery in a parser for Pascal. The BC(1,1) limitation on the grammar caused problems in the definition of Pascal, which where alleviated by permitting the parse table generator to rewrite the grammar if necessary.

Lang describes a method for parsing sentences containing an arbitrary number of unknown parts of unknown length [12]. The parser produces a finite representation of all possible parses (often infinite in number) that could account for the missing parts. The implementation of this method is

based on Earley parsing [5], as is the Tomita algorithm we use in our own substring parser. The basic idea of Lang's method is that "in the precence of the unknown subsequence $*$, scanning transitions may be applied any number of times to the same computation thread, without shifting the input stream." This process terminates, as parsers in the same state are joined and the number of states is finite. This method is very elegant and powerful, and can be used as a substring parser (by providing it with the string "$*s*$"). We will not use it, however, as it is more general than what we need. Whether it would be efficient enough for interactive purposes is unclear.

Snelting presents a technique to complete the right-hand side of unfinished sentences [16] (also see Section 5.2).

4 Substring Recognition

4.1 Tomita parsing

We base the implementation of our substring parser on Tomita's algorithm. This algorithm runs several simple LR parsers in parallel. It starts as a single LR parser, but, if it encounters a conflict in the parse table, it splits in as many parsers as there are conflicting possibilities. These independently running simple parsers are fully determined by their parse stack. When two parsers have the same state on top of their stack, they are joined in a single parser with a forked stack. A reduce action which goes back over a fork in a parse stack, splits the corresponding parser again into two separate parsers. If a parser hits an error entry in the parse table, it is killed by removing it from the set of active parsers. The possibility to run several parsers in parallel makes the Tomita algorithm very well suited for substring parsing.

For a full description of the Tomita parsing algorithm we refer to Tomita [17], to Nozohoor-Farshi who corrected an error in the algorithm concerning ϵ-productions [13], or to Rekers who extended the algorithm to the full class of context-free grammars by including cyclic grammars[1] [14]. For a detailed explanation of LR parsing [2, ch. 4.7] is recommended.

[1]Grammars in which $A \stackrel{+}{\Longrightarrow} A$ is a possible derivation

4.2 The grammar

The grammar for which our substring recognition algorithm works should be reduced in such a way that it does not contain non-terminals that cannot produce any terminal string or ϵ. These non-terminals can be identified easily, and all rules in which they appear should be removed from the grammar. This clean-up operation does not affect the language recognized. [9, p. 73-76]

Useless symbols and unreachable rules do not influence substring parsing as these are ignored by the parse table generator. This is due to the fact that LR parse tables are generated top-down, starting with the start symbol of the grammar, and that useless symbols and unreachable rules are, by definition, unreachable from the start symbol.

4.3 The algorithm

If we have to determine whether a string $s_0 \cdots s_n$ is a substring of a sentence in a language L, we start the substring recognition process by generating, for each state directly reachable under s_0, a parser with this state on its stack. These parsers will process $s_1 \cdots s_n$.

We will show how an individual parser processes an action, but we will not discuss the management of the different parsers, as this is done in the same way as in ordinary Tomita parsing. The parser obtains an action from the parse table with the state on top of its stack and with input symbol s_k. This can be a *shift*, *error* or *reduce*-action, and is processed in the following manner:

- A (shift *state'*)-action is processed as in normal parsing: *state'* is pushed on the stack and the parser is ready to process s_{k+1}.
- An (error)-action removes the parser from the set of active parsers.
- A (reduce $A ::= \alpha\beta$)-action is processed as follows:
  - If there are at least $|\alpha\beta| + 1$ entries on the parse stack the reduce action is performed as in normal parsing: $|\alpha\beta|$ entries are popped off the stack, and the parse table is consulted, with the state remaining on top of the stack and A, to obtain a state to push on the stack again. The parser is now ready to continue the processing of s_k.

- If there are only $|\beta|$ entries on the stack, only β has been recognized of $A ::= \alpha\beta$; α lies before s_0 and should produce (a part of) a prefix of s_0. This is possible, as all non-terminals in α can produce some terminal string, and all terminals in α trivially do. So the reduction $A ::= \alpha\beta$ may be performed. The states which can be reached directly by a transition under A are the states where parsing may continue. For each of these valid states a new parser is started with that state on the stack. These parsers all proceed to process s_k.
- If there are exactly $|\alpha\beta|$ entries on the stack, $s_0 \cdots s_{k-1}$ reduces to $\alpha\beta$, but the context in which A is to be used is unknown. This is handled in the same way as the previous case.

If there are no parsers left alive after the processing of s_n, the substring parser fails. If there are parsers left, these are currently recognizing rules $A ::= \alpha\beta$, of which (a part of) α has been recognized. As every β can produce some terminal string, these rules can all be finished. This means that the substring parser succeeds if there are parsers remaining after the processing of s_n.

4.4 The parse table generator

The substring parser is controlled by the same parse table as our ordinary parser. To generate this parse table we use an extended version of the lazy and incremental parser generator IPG [10]. The extension concerns the need of the substring parser to know all states which can be reached by a transition under a given symbol. This function needs global information about the parse table, which means that the whole parse table must be known. As a consequence, the lazy aspect of IPG cannot be exploited here and the parse table is always fully expanded. The expanded parse table can also be used by the ordinary parser, of course.

5 Substring Parsing

We extend the substring recognizer into a substring parser by generating parse trees for substrings. The possible parse trees for a substring s are the parse trees of all sentences vsw for which $vsw \in L$ holds. To limit the number of completions we allow v and w to consist both of *terminals* and *non-terminals*, and we generate a parse tree,

START ::= *Stat*
START ::= *Exp*
Stat ::= if *Exp* then *Stat*
Stat ::= if *Exp* then *Stat* else *Stat*
Stat ::= *Id* := *Exp*
Exp ::= *Id*
Exp ::= *Int*
Exp ::= *Exp* + *Exp*
Exp ::= *Exp* * *Exp*
Exp ::= (*Exp*)

Figure 1: A grammar

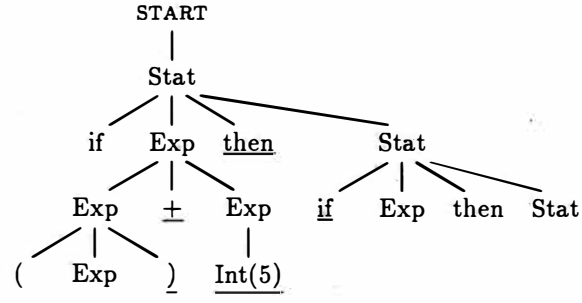


Figure 2: A completion of ") + 5 then if"

corresponding to a sentential form $\sigma_1 s \sigma_2$, only when the frontier of each of its *subtrees* contains at least one symbol of s; i.e., we do not generate subtrees whose frontier lies entirely within σ_1 or σ_2. The trees that we generate are the most general trees, as it is not possible to replace any of their subtrees by a non-terminal such that the frontier still contains s as a substring. Even so, the number of completions can still be infinite. In Section 5.2 we will discuss how to limit this number still further.

For the grammar of Figure 1 and the string ") + 5 then if", a possible completion is the sentential form

$$\underbrace{\texttt{if (} \mathit{Exp} \texttt{) + 5 then if}}_{\sigma_1} \quad \underbrace{\mathit{Exp}}_{s} \quad \underbrace{\texttt{then } \mathit{Stat}}_{\sigma_2}$$

whose parse tree is given in Figure 2. To distinguish the leaves of s from those of σ_1 and σ_2, the former are underlined.

5.1 Generating the completions of a substring

LR parsers generate parts of parse trees during a reduction step. On reducing $A ::= \alpha$, the parse stack contains the subtrees created for α. These

are assembled in a new node of type A and the subtree created in this way is pushed on the stack. In the substring parser ordinary reductions are treated in the same way.

If the rule $A ::= \alpha\beta$ is reduced with only nodes for β on the stack, however, additional nodes are created for α. In this way, the parse trees for the possible prefixes of s are created.

Parse trees for postfixes of s are created in the same way: after processing s the parser has to finish all rules which are in the process of being recognized. These are the rules in the kernel of the current state of the parser. If only α has been seen from a rule $A ::= \alpha\beta$, the rule is reduced and additional nodes are created for β. It can even be the case that only β has been recognized from a rule $A ::= \alpha\beta\gamma$, and that nodes must be created for both α and γ.

5.2 Further reduction of the number of possible completions

By producing only parse trees that are most general, the number of possible completions is reduced, but it is often still too large and not even always finite. We propose the following rules to limit this number still further:

1. The parse trees generated are kept as compact as possible by disallowing derivations of the form $A \stackrel{+}{\Longrightarrow} \alpha A$, $A \stackrel{+}{\Longrightarrow} \alpha A \beta$, and $A \stackrel{+}{\Longrightarrow} A\beta$, where only A has actually been recognized and all elements of α and β would produce elements in σ_1 or σ_2. Clearly, such derivations can be repeated infinitely often. They are undesirable as they only enlarge σ_1 or σ_2.

   For example, the substring "`) + 5 then if`" also has a possible completion

   $$\underbrace{\texttt{if } \mathit{Exp} \texttt{ + (} \mathit{Exp}}_{\sigma_1} \underbrace{\texttt{) + 5 then if}}_{s} \underbrace{\mathit{Exp} \texttt{ then } \mathit{Stat}}_{\sigma_2}$$

   whose parse tree is given in Figure 3. In this tree a subtree for the rule $Exp ::= Exp + Exp$ has been inserted in the prefix.

2. The number of possible sentential forms for which parse trees are generated is now finite, but these can still have infinitely many parse trees as the grammar may be cyclic. Rekers describes how to parse and generate *parse graphs* for cyclic grammars [14]. The cycles generated in this graph can be removed by his routine *remove-cycles*. This results in a finite number of most general completions.

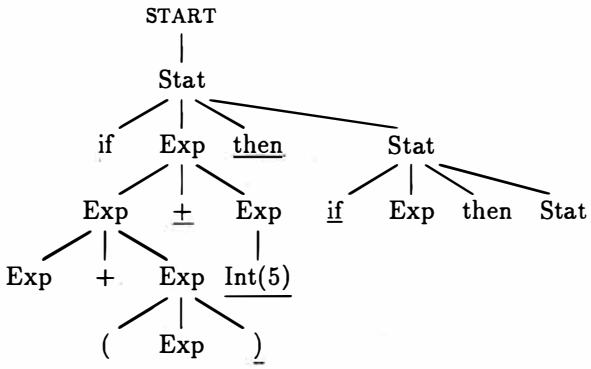


Figure 3: Another possible completion of "`) + 5 then if`"

3. In the generation of the postfixes of s a choice can be made for the "simplest" completion. That is, if a substring can be completed according to both $A ::= \alpha\beta$ and $A ::= \alpha\gamma$, and $|\beta| < |\gamma|$, we prefer $A ::= \alpha\beta$. In the example of Figure 2 this rule forbids the choice of the "if-then-else" rule, as the "if-then" rule already applies. Snelting's rule "*prefer reduce items over shift items*" [16] is similar to ours. It can also be formulated as: if completion according to both $A ::= \alpha$ and $B ::= \alpha\gamma$ $(\gamma \neq \epsilon)$ is possible, then prefer $A ::= \alpha$. We consider our rule more appropriate, as we take the case of β being non-empty but shorter than γ into account as well, and we only make the choice if the two rules reduce to the same non-terminal. Otherwise, the rule $A ::= \alpha$ might be preferred over $B ::= \alpha\gamma$, whereas the environment in which the substring is completed needs a tree of type B.

6 Measurements

Our first measurement compares the substring recognizer with the Tomita recognizer from which it was derived to learn the additional costs of substring parsing.[1]

We have taken a grammar of about twenty rules and sentences of increasing length. These were parsed by the Tomita recognizer first. The resulting parse times are indicated in Figure 4 with a "•". Next, the same strings minus a randomly chosen prefix were given to the substring parser.

[1]The measurements were performed on a SUN Sparc station. The programs were written in Lisp. The time used by the lexical scanner has not been taken into account.

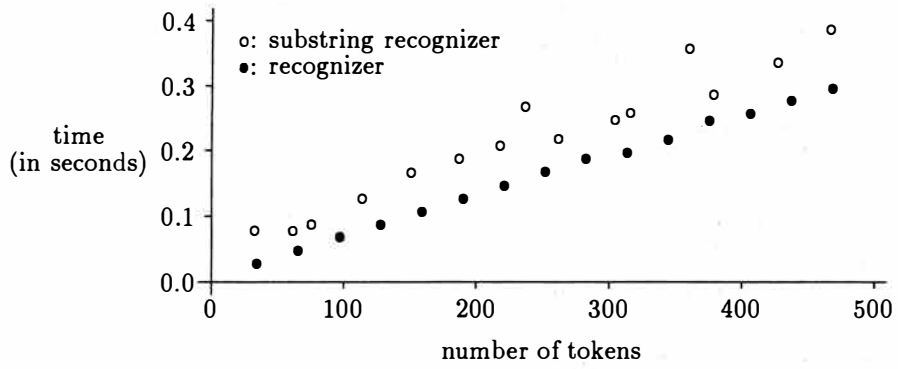


Figure 4: Comparison of the substring recognizer with an ordinary one

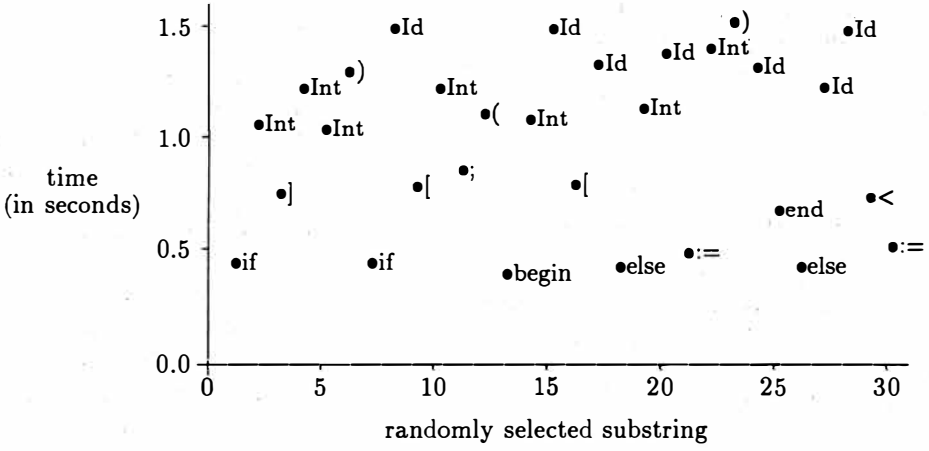


Figure 5: Time needed by the substring parser on Pascal sentences of 100 tokens

The required times are indicated in Figure 4 with a "o".

It turns out that the substring parser has a moderate overhead with respect to the normal parser. This overhead can be interpreted as the time needed for the substring parser to get on the "right track". As Figure 5 shows, the variations in this overhead are caused by the random cutting of the string. For some strings it takes longer than for others to determine of which language construct it can be a substring. The larger the grammar is, the more alternatives are available and therefore the higher the variation.

In Figure 5 we compared the time taken by the substring parser on 30 randomly chosen parts of Pascal sentences of 100 tokens. The dots indicate the amount of time needed and they are attributed with the first symbol of the substring. These measurements show that sentences starting with a token that can appear in many differents contexts, like "`Id`" or "`)`", take more time to recognize than sentences starting with a disambiguating token like "`:=`" or "`else`".

7 Conclusions

The adaptation of the Tomita algorithm to substring parsing results in a very elegant and powerful algorithm. The main advantage of the fact that it accepts general context-free grammars and uses ordinary LR parse tables is that substring parsing can now be applied in a very general manner, instead of only to carefully written grammars and at the cost of an extra generation phase.

Substring parsing is slower than ordinary parsing, but this will not be a serious drawback for its application as an error recovery technique or as a completion tool. The use of the substring parser in incremental parsing, however, has to be investigated further.

Acknowledgments

We would like to thank Nigel Horspool, who suggested to extend our implementation of the Tomita algorithm to substring parsing. Two years

after this discussion we finally saw the need for such a technique and started a serious investigation. Next, we are grateful to Paul Hendriks who pointed out a valuable simplification in the treatment of incomplete reductions in the substring parser, and to Jan Heering for his careful reading of earlier versions of this paper.

References

[1] R. Agrawal and K.D. Detro. An efficient incremental LR parser for grammars with epsilon productions. *Acta Informatica*, 19:369–376, 1983.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] A. Celentano. Incremental LR parsers. *Acta Informatica*, 10:307–321, 1978.

[4] G.V. Cormack. An LR substring parser for noncorrecting syntax error recovery. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 161–169, 1989. Appeared as *SIGPLAN Notices* 24(7).

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] R.W. Floyd. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67, 1964.

[7] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, 1979.

[8] C. Ghezzi and D. Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–579, 1980.

[9] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[10] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 179–191, 1989. Appeared as *SIGPLAN Notices* 24(7).

[11] J.W.C. Koorn. GSE: A generic text and structure editor. Programming Research Group, University of Amsterdam, to appear.

[12] B. Lang. Parsing incomplete sentences. In *Proceedings of the Twelfth International Conference on Computational Linguistics*, pages 365–371, Budapest, 1988. Association for Computational Linguistics.

[13] R. Nozohoor-Farshi. Handling of ill-designed grammars in Tomita's parsing algorithm. In *Proceedings of the International Parsing Workshop '89*, pages 182–192, 1989.

[14] J. Rekers. Parsing for cyclic grammars. Centrum voor Wiskunde en Informatica (CWI), Amsterdam, in preparation.

[15] H. Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, 1985.

[16] G. Snelting. How to build LR parsers which accept incomplete input. *SIGPLAN Notices*, 25(4):51–58, 1990.

[17] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic Publishers, 1985.

[18] D. Yeh. On incremental shift-reduce parsing. *BIT*, 23(1):36–48, 1983.

PARSING WITH RELATIONAL UNIFICATION GRAMMARS

Kent Wittenburg
Bellcore Visiting Researcher
Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759
Arpanet: Kent@mcc.com
Phone: (512)338-3626
Fax: (512)338-3600

ABSTRACT

In this paper we present a unification-based grammar formalism and parsing algorithm for the purposes of defining and processing non-concatenative languages. In order to encompass languages that are characterized by relations beyond simple string concatenation, we introduce relational constraints into a linguistically-based unification grammar formalism and extend bottom-up chart parsing methods. This work is currently being applied in the interpretation of hand-sketched mathematical expressions and structured flowcharts on notebook computers and interactive worksurfaces.

1. INTRODUCTION

In the MCC Interactive Work Surface Project, we have been applying a language perspective to the problem of connecting meaning to graphical and sketched media on both the input and the output side of human-machine interfaces. The technology is initially being applied to the problem of recognition of hand-sketched input through the "electronic paper" interface of notebook computers and worksurfaces (Avery 1988; Martin et al. 1990). Our first applications are interpreters for math expressions and structured flowcharts. Subsequent applications will include interpretation of sketched designs (e.g., engineering or architectural layouts or plats) in such a way that the semantic information can be made available for subsequent database update and querying, intelligent advising, creation of dynamic prototypes, etc. On the output side, we expect that the inverse connection of underlying data to a visual vocabulary will enable easy-to-use tools for connecting the semantics of underlying data to dynamic graphical displays.

Figure 1-1 shows a visualization of a derivation in two-dimensional space. Such derivations can be produced by grammars which describe languages whose sentences are objects situated in a two-dimensional space as long as the grammars can specify relational, in the most obvious case positional, constraints among the objects. Such constraints, and their resolution, are beyond the capacity of structurally-based unification grammars and parsing methods developed for languages of strings. The purpose of this paper is to present the framework of Relational Unification Grammar (RUG), which is capable of describing such languages, and then to extend bottom-up chart-parsing methods to work with these grammars. The algorithm presented here is motivated by the need to process input incrementally. We expect to derive benefit in our application domains from processing each symbol in the order in which it is created by a user. Such a parser allows for suggesting possible continuations for the user as well as determining correctness of the input so far. Temporal ordering imposes significant demands on our parser since we cannot enumerate the input based on spatial considerations, for example, a top-down left-to-right traversal. Such a normalization of ordering has usually been assumed in previous applications of grammar-based parsing to visual domains (e.g., Tomita 1989; Chang 1988).

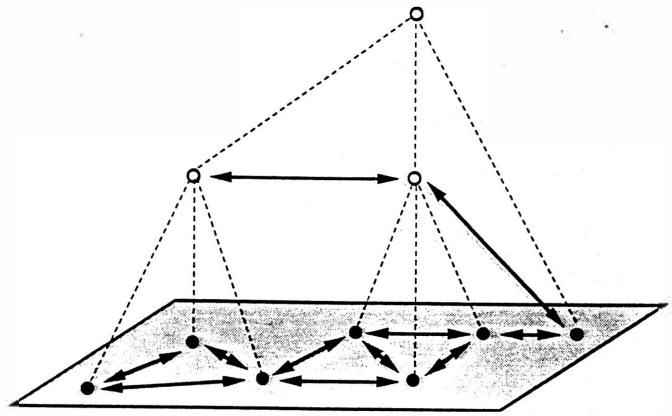


Figure 1-1: Derivation in 2D space

2. GRAMMARS

Parallel to Helm and Marriott (1990), who are investigating visual languages in the logic programming tradition, we adopt a unification-based grammar formalism that is augmented with constraints necessary to incorporate relations beyond string concatenation into the declarative specification of a language. Unification itself then must be expanded to incorporate some form of constraint solving, an area of active research in logic programming.

Our approach is to extend the family of PATR unification-based grammar formalisms (Shieber 1986, 1989). Instead of strings, we assume the terminals of our grammar to be what we will call icons, objects that are associated with a set of attributes such as <X,Y> coordinates, extent, and color, each of whose value ranges is finite. The rules of the grammar, besides specifying a variety of syntactic and semantic constraints for use in deriving sentences of the language, also specify relational constraints among icons that may require arbitrary computation to determine satisfaction.

Although we will not attempt to give a rigorous definition of the unification basis of our grammars here, it will nevertheless be useful to note properties of some of the attributes, values, and relational constraints appearing in the grammar. Besides the customary PATR grammar machinery consisting of a vocabulary of attribute labels L and constant values C, lexical and nonlexical productions P, and a start category (see Shieber 1989), a relational unification grammar RUG is distinguished by the tuple (N,Σ,I), where N is a finite set of (nonterminal) icon type symbols, Σ is a finite set of (terminal) icon type symbols, I is an infinite set of spatially located icons each of which has a type $\in N \cup \Sigma$, and R is a finite set of relations in I.

The rules of the grammar contain the following elements, whose left-hand sides we will consider unordered for the time being.

$$\text{Head } \text{Arg}_1 \ \ldots \ \text{Arg}_n \rightarrow \text{Result}$$

Since we are focusing on analysis here rather than generation, the arrow in the rule skeleton is interpreted as "reduces to" rather than "rewrites as". Each rule must have a head and a result, and there may be zero or more arguments. Although there is nothing essential from a formal point of view about our use of the functional terms *head*, *argument*, and *result* in rules, it is a convention to guide grammar construction that we find perspicuous.

Each of the elements of the production has at least the following structural constraints:

$$\text{syntax} \in N \cup \Sigma$$
$$\text{icon} \in I$$

The *syntax* attribute must take as value elements from the set $N \cup \Sigma$. Although in fact we allow for syntactic characterizations to be arbitrarily complex, we need a designated feature somewhere in the structure to be able to instantiate and refer to the types of icons associated with both terminal and nonterminal symbols. Here we will use the *syntax* attribute for this purpose. In addition, each of these rule elements has an *icon* attribute whose values are taken from I. In practice, the *icon* value may be a unique name for an icon instance.

Additionally, every rule that is not unary is required to have a set of relational constraints with certain additional conditions. Let us turn to an example in order to clarify the use of relational constraints. Following is an example of a rule from the domain of mathematics expressions. It is a rule that forms vertical infix expressions such as fractions, assigning an appropriate semantics for evaluation of the expression. The syntactic and semantic structural constraints appear first followed by the relational constraints.

Rule 1 Vertical infixation:

$$\text{Head} \quad \text{Arg}_1 \quad \text{Arg}_2 \quad \rightarrow \quad \text{Result}$$

$$\langle \text{Head icon} \rangle = X$$
$$\langle \text{Arg}_1 \text{ icon} \rangle = Y$$
$$\langle \text{Arg}_2 \text{ icon} \rangle = Z$$
$$\langle \text{Head syntax} \rangle = \text{Vert-infix-op}$$
$$\langle \text{Arg}_1 \text{ syntax} \rangle = \text{Formula}$$
$$\langle \text{Arg}_2 \text{ syntax} \rangle = \text{Formula}$$
$$\langle \text{Result syntax} \rangle = \text{Formula}$$
$$\langle \text{Head sem} \rangle = \langle \text{Result sem pred} \rangle$$
$$\langle \text{Arg}_1 \text{ sem} \rangle = \langle \text{Result sem arg1} \rangle$$
$$\langle \text{Arg}_2 \text{ sem} \rangle = \langle \text{Result sem arg2} \rangle$$

$$\langle \text{Result icon} \rangle = \text{composition}(X\ Y\ Z)$$
$$\text{above}(Y\ X)$$
$$\text{below}(Z\ X)$$
$$\text{wider-than}(X\ Y)$$
$$\text{wider-than}(X\ Z)$$

The first of the relational constraints, which involves *composition*, defines the icon of the rule mother as a function of the icons of the rule daughters. In practice, this relation may involve summation of the bounding boxes of the daughter icons for domains such as math expressions, or concatenation of line segments in diagram domains. The other relations impose positional and size constraints on the icons involved in the production. Suitable definitions of *above* and *below* in the math domain incorporate adjacency as well as position. The particulars of such relations will differ across grammars and domains.

In anticipation of the bottom-up parsing algorithm which we will present shortly, there is an additional requirement which we will impose on the form of grammar productions and their relational constraints. Given that there are no positional constraints implied by the rule skeletons, it is useful for the parser to be driven by appropriate relational constraints from individual rules for its basic rule matching operations. Thus we distinguish the class of relations that drive the matching action of the parser from those that operationally serve as constraints on proposed matches. Positional constraints such as *above* are more appropriate for driving parsing than size constraints such as *wider-than*. We will assume that each grammar distinguishes a class of positional constraints for this purpose.[1] Furthermore, we will refer to the maximal relational domain (R domain) over which positional relations hold. Many grammars will restrict their positional constraints to adjacent elements--in this case, adacency defines the R domain for that grammar.

Our requirement on rule wellformedness is that there be some ordering of the daughter elements in productions as follows:

> **Condition 1:** An ordering of rule daughter elements is well-formed iff for every element but the first, a positional constraint exists between that element and an element appearing earlier in the ordering.

An intuitive understanding of the reason for Condition 1 can be reached by considering the ordering <Arg_1, Arg_2, Head>, corresponding to the order <numerator, denominator, divide-line> in a fraction expression, from Rule 1. Suppose the parser has matched the numerator element and is in the position of seeking candidates for its next match, the denominator element. Since there are no positional constraints in the rule that involve the icon associated with the instantiated numerator, the parser has no way to constrain the candidates for its next match. One would of course like to confine the search to only those objects which meet appropriate positional constraints from the grammar. On the face of it, the parser would have to consider every icon in the space as a possible instantiation of the denominator term in our example and could not rule any of these branches out on the basis of relational constraints until the divide-line had been matched.

[1]It will also simplify our exposition slightly if we assume that the icon variable for every argument appears as the domain term of at least one positional constraint. For this reason, we use both *above* and *below* in Rule 1, even though the same constraints could be stated with just one of these relations.

Fortunately, this condition on the form of rules can be determined off-line, and we assume that a particular ordering of the elements of a rule is prespecified that meets this condition. For the purposes of this paper, we will assume that daughter elements of rules are to be matched in the order in which they are given in the rule definitions, implying that rules will be matched head-first.[2]

As a final note, lexical productions are defined traditionally as in PATR grammars with the difference that instead of strings, the terminal vocabulary is taken from the set of icon type symbols Σ. That is, lexical entries are pairs of the form $\langle\sigma, \Phi\rangle$, where σ is a member of the set of terminal icon symbols Σ and Φ is an RUG formula containing structural attributes found in the individual elements of rules. For example, here is a possible lexical entry for a line representing division:

```
horizontal-line:

    <syntax> = vert-infix-op
    <icon> = X
    <sem> = divide
```

Note the that *icon* attribute is uninstantiated in the lexicon. It will be instantiated with an actual icon instance (or a reference to one) during lexical lookup.

An example of a simple grammar may be found in Section 4, where we show a parse trace.

3. PARSING

In this section we give an account of a data-driven, tabular parsing algorithm that uses the grammar formalism described above. The algorithm we describe is technically a recognition algorithm, though it is easy to extend it to a parsing algorithm through the standard methods available in the literature (Aho and Ullman 1972). Tabular parsing methods (e.g., Earley 1970) and closely related chart parsing methods (Kaplan 1973; Kay 1980) have in common the use of a grammar table (or chart) that stores all complete and partially matched constituents, indexing them

[2]Although the parsing algorithm we discuss matches the elements of rules deterministically, algorithms such as Satta and Stock's (Satta and Stock 1989), which match rules in variable orders starting with the head, could be adapted to these grammars if any and all orderings meet Condition 1. One would, however, have to add the overhead necessary to check for the possibility of achieving the same rule match in more than one order.

to spans of the input string. The tables are used both to merge equivalent constituents over the same input into a single entry, thus avoiding combinatorics, and also to propose candidates for rule applications, given that adjacent entries in the tables are tied directly to adjacent substrings in the input.

Two approaches have been employed previously to apply tabular parsers in visual language domains. The first is to convert visual input data into a one-dimensional string form and use conventional string-based parsing methods. According to Fu (1974), the linear-conversion approaches have "not been very effective in describing two- or three-dimensional patterns". The second approach is to extend conventional parsing tables to directly represent regions over a spatial domain rather than spans over an input string. Tomita (1989) has extended the Earley algorithm and his own LR methods in this manner. Such an option ties a parser to the particulars of the spatial concatenation operations allowed in the grammar since the makeup of the table itself will be affected by the set of relations permitted in the visual space.

In contrast, our approach is to redistribute the functions expected from a parsing table across two modules. One, discussed in detail here, incorporates the parsing table and its constituent entries. From these data structures one can determine the input which a constituent dominates in order to check for equivalent table entries and successful output; however, one cannot from these structures alone determine the candidates for extending constituent coverage through rule applications. The module called the spatial relations analyzer, which keeps its own set of data structures, is necessary to discover new icon candidates for incorporating into rule applications. We hope that this overall conceptual design will become clear in the descriptions and examples which follow.

We assume an unordered set of spatially located icons as input to the parser. The correctness and completeness of the algorithm will not be affected by any temporal ordering of the input, but, for that reason, we can process the icons incrementally, in the order in which they appear through the interface.

> **Definition 1:** A *cover*, defined with respect to entries (partial or complete grammatical constituents) in the parse table, is the subset of icons in the input set that an entry dominates.

Covers are necessary for determining equivalence of constituents and success for the parse. The goal of parsing will be to produce any and all consitutents covering the initial input set that are labeled with the start symbol of the grammar.

Note that a cover need not be contiguous in a temporally determined input sequence. However, contiguity of a cover in the two-dimensional space will be enforced to the extent that the grammar uses spatial relations that subsume adjacency.

> **Definition 2:** A *category* is defined to be a PATR formula that is either a (partially) instantiated production as defined in Section 2 or else a PATR formula with instantiated features *syntax* and *icon*.

Categories are (partial) instantiations of rules, rule results, or lexical categories. In the algorithm descriptions which follow, we will assume the convention of referring to relevant features of categories with the notation *[head, arg_1...arg_n, result]* in the case of partial rule instantiations and *[syntax, icon]* in the case of rule result or lexical instiantions. We will also refer to individual rule elements at times with the convention *[syntax, icon, rels]*, where rels is a sorting of all relational constraints in the rule that contain the element's icon in either the domain or range term.

> **Definition 3:** A *state* is defined to be a triple [category, next-arg, cover], where next-arg refers to an arg i...j of category, possibly empty.

States are the parser's representation of a constituent. States are said to be *active* if next-arg is nonempty, implying that the category is an incomplete constituent, or *inactive* if next-arg is empty, implying that the category is a complete constituent. Active states will have partial rule instantiations as categories; inactive states will have instantiations of rule results or lexical items.

> **Definition 4:** A *trigger*, defined with respect to active states, is any (instantiated) icon appearing in the range term of a positional constraint whose domain term is the next-arg's icon variable.

That is, consider an active state that fits the category schema

$$\text{Head} \ldots \text{Arg}_j \ldots \rightarrow \text{Result}$$

$$\begin{array}{l} \langle \text{Head icon} \rangle = \text{Icon}_1 \\ \langle \text{Arg}_j \text{ icon} \rangle = X \\ \quad --- \\ (\text{Rel}_1\ X\ \text{Icon}_1) \end{array}$$

and whose next arg is Arg_j. $Icon_1$ will be a trigger for this active state since it appears in the range term of a positional constraint together with the next-arg's icon as domain. Note that we do define triggers to be instantiated icon instances, not icon variables.

In some cases there may be more than one trigger icon defined for an active state. Consider an active state whose category matches the following schema

```
Head ... Arg_j ... Arg_k ... → Result
  <Head icon> = Icon_1
  <Arg_j icon> = Icon_2
  <Arg_k icon> = X
       ---
  (Rel_h X Icon_1)
  (Rel_i X Icon_2)
```

and whose next-arg is Arg_k. Both $Icon_1$ and $Icon_2$ are triggers for this state. In such a situation the parsing algorithm, which uses triggers to index active states in the parse table, needs only one trigger. We arbitrarily choose among them.

Before turning to the parsing algorithm itself, we need one final definition. Lexical lookup, which produces states with instantiated categories associated with incoming icons, is defined next.

Definition 5: The function Lex(ical lookup), from the set of icons to a set of pairs consisting of a state and its icon index, is defined as follows:

```
Lex(X)={(s=[category, nil, {X}], i=X) |
  category = a lexical entry indexed by
  icon-type(X) and whose <icon> is
  unified with X}
```

This function represents lexical lookup and state instantiation. Given an icon, it uses the icon's type symbol to consult the lexicon. With the set of categories the lexicon produces, it then initializes the data structures for placing inactive states onto the parse table. In so doing, it unifies the icon itself with the icon variable of the category. The cover of the category will be the unary set consisting of the icon again. The index is an icon which will be used to index the state in the parse table. In the algorithm presented here, all lexically instantiated states will be inactive--the index for inactive states will be the icon for which the state represents a complete constituent.

Algorithm 1 *Main loop*

Assume an input set of spatially located icons W and an agenda set A, initially empty. Develop a table T whose entries are state sets indexed by icons.

```
while A nonempty or there exist icons
remaining to be processed in W do:
 choose one of the two following actions:
    for some icon X in W do              (1)
        for each pair (s,i) in Lex(X) do
            add (s, i) to set A
    extract any pair                     (2)
      (s=[category, next-arg, cover], i)
        from the set A;
        insert s in table T_i;
        apply each of the following
        procedures, in any order, to s:
            propose(s)
            expand(s)
            complete(s)
end while;
if there exists an s =
[category, next-arg, cover] in T such that
cover = W,  next-arg = empty, and the label
of category = start,
   then succeed;
   else fail.
```

The basic algorithm chooses arbitrarily among two actions as long as there are remaining data to do either action. Action (1) processes a single arbitrary icon from the input set. It creates state-index pairs to be placed in set A--this set, in chart parsing, corresponds to an agenda of pending actions. Action (2) chooses an arbitrary state-index pair from the agenda. It inserts the state into the parse table and then generates more pairs for the agenda by applying the three procedures *propose*, *expand*, and *complete* in any order. The indices for states in the parse table are icons. As will become evident in the procedures that follow, the index for active states is a trigger icon for that state; for inactive states, it is the icon associated with the highest dominating nonterminal.

Procedure 1 *Propose*

```
If state s=[category, nil, cover] is inactive,
then for every production p in P
   such that the category of s
   unifies with head element of p,
     create a new pair
     (s'=[category', next-arg, cover'], index)
     as follows:
       if there are no arguments in p
          then category' ::= result of p
               next-arg ::= nil
               cover' ::= cover
               index ::= icon of category;
          else category' ::= p
               next-arg ::= arg_1 of p
               cover' ::= cover
               index ::= a trigger icon of s';
     add pair to A unless an equivalent pair
     already exists.
```

The *propose* procedure applies to inactive states. It proposes new constituents through trying to unify the category of an inactive state against the head terms of the rule set. Successful unifications will result in new states that will be active or inactive depending on whether the rule is unary or not. Active states have a next-arg pointing to the first argument of the rule to be matched; inactive states have a null next-arg. The index of a new state will be the icon associated with the

newly unified category if the state is inactive, or a trigger icon if the state is active.

The condition that new states are added only if there is not an equivalent state already in A is a necessary (but not sufficient) condition for keeping the algorithm polynomial. This is a familiar move for all On^3 bounded context-free parsing algorithms. We will not elaborate here on questions of computational complexity, but suffice it to say we assume a definition of equivalence of <state, index> pairs--they are equivalent if their covers and indices are equal and if their categories and advancement are equivalent. A parsing algorithm, rather than just a recognition algorithm such as the one we are discussing here, would need to keep track of these equivalent states in order to recover the full set of parse trees.[3]

The fact that this algorithm proposes new rules for matching only when head elements of rules are discovered is part of the formula for making this algorithm "head-driven". It would be possible to use the predictive power of the partially matched headed constituents to filter out useless argument constituents. In the basic data-driven algorithm we discuss here, however, we do not actually make use of such top-down predictive machinery.

Procedure 2 *Expand*

```
If state s=[category,next-arg,cover]
   with next-arg=[syntax,Y,rels] is active,
then for some trigger icon X in a relation
  (rel Y X) in rels,                          (1)
  for every icon Z in the space such that
  (rel Z X)=True,                             (2)
      then for every inactive state
      s'=[category', nil, cover'] indexed
      by Z,
      if category' unifies with next-arg      (3)
         then (advance s s'),
         add resulting pair (s'' i) to set A
         unless an equivalent state exists.
```

The *expand* procedure is used to advance an active state across its next argument by finding inactive states that match the constraints of that argument as specified in the partially instantiated rule. Finding the candidate inactive states is the crux of the matter. They must (a) be associated with icons that meet the relational constraints of the argument, and (b) have categories that unify with the structural constaints of the rule's next argument. We use the partially instantiated relational constraints, relying on our spatial relations module, as a means of finding the icons that meet the spatial requirements. This particular feature of the algorithm is necessary given that we are not relying on our parse table to provide us with, say, adjacent icons.

The procedure begins with a trigger icon for an active state. Recall Definition 4 for triggers; line (1) of the procedure essentially restates it. Given a trigger icon, line (2) looks in the physical space for any icons in the triggering relation. The ones it finds will then be candidates for the icon to be associated with the next-arg. The remaining steps find any inactive states associated with the icon in question and then check that the structural features of these states as well as any remaining relational constraints are consistent with the rule's requirements. (Both conditions are implied by the unification step in line (3).) Any states that satisfy these conditions will be combined with the orginal active state, producing a new state that covers more territory.

Procedure 3 *Complete*

```
If state
   s=[category=[syntax, Y], nil, cover]
   is inactive,
then for every icon X falling within the
   local R domain w.r.t. icon Y,          (1)
   for every active state
   s'=[category',next-arg,cover'] that is
   indexed by X as trigger,               (2)
     if next-arg of s' unifies with category,
          then (advance s s'),
          add resulting pair (s'' i) to set A
          unless an equivalent state exists.
```

The *complete* procedure is defined with respect to inactive states. The basic operation is to look for active states for which this new inactive state can serve as a next argument, and then advance any such active states with respect to the inactive state. It operates just like the *expand* procedure once the candidate states are found. The differences lie in how one finds candidate active states given an inactive state, rather than the reverse.

As is indicated in line (1), the procedure depends on a notion of locality in the space in order to find the initial set of icons that is used to begin the search. If the R domain were characterized by adjacency, the procedure would map over each of the icons that were adjacent to the icon associated with the new inactive state. We do not, however, rule out the possibility that the locality of spatial relations may be defined otherwise.

Line (2) then consults the parsing table to find active states indexed by the locally related icons. Recall that active states are indexed by trigger icons. Thus these states will be the ones which the original inactive state may combine with. Further steps are the same as in *expand*.

[3]The issue of equivalence and state merging is nontrivial for unification grammars. See Shieber (1985).

Procedure 4 *Advance*

```
Given active state s=[category,next-arg,cover]
  with next-arg=[syntax, Y,(rel X)]
  and inactive state
  s'=[category', nil, cover'],

create a state s'' with index i as follows:

 case1: if category has no further arguments,
  then create a pair
  (s''=[category'', nil, cover''], i)
  where category'' ::= result of category,
        cover'' ::= cover' Union cover.
        i ::= icon of category''.

 case2: if category has further arguments,
  then create a pair
  (s''=[category'', next-arg', cover''], i)
  where category'' ::= category,
        next-arg' ::= next-arg + 1,
        cover'' ::= cover' Union cover,
        i ::= icon trigger for s''.
```

Advance takes an active state s and an inactive state s' which has already been unified as the next-arg for s, and it returns a new state/index pair. The new state resulting from advancement will be either inactive or active, depending on whether the final argument of the active state has been matched or not. The creation of an inactive state, shown in case1, sets the new state's category to the result-category of the active state. Its index will be the icon newly formed from the composition relation that holds between the icon of the result and the icons of the rule daughters. The creation of active states involves an advancement of the next-arg pointer. These states are indexed by a trigger icon. In both cases, the cover for the new state will be the union of the covers of the original states.

4. EXAMPLE

Here we give a example of a grammar for simple fractions and a parse trace of the bottom-up algorithm described above. Rule 1 is repeated for convenience. The trace will refer to the rules and lexical entries by number and omit the details of the internal rule elements. When nil appears in the next-arg position of a state, it is an indication that the category of the state corresponds to the instantiated result element of completed rules or the categories of lexical entries.

Rules

1 Vertical infixation:

Head Arg_1 Arg_2 → Result

<Head icon> = X
<Arg_1 icon> = Y
<Arg_2 icon> = Z
<Head syntax> = Vert-infix-op
<Arg_1 syntax> = Formula
<Arg_2 syntax> = Formula
<Result syntax> = Formula
<Head sem> = <Result sem pred>
<Arg_1 sem> = <Result sem arg1>
<Arg_2 sem> = <Result sem arg2>

<Result icon> = composition(X Y Z)
above(Y X)
below(Z X)
wider-than(X Y)
wider-than(X Z)

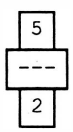
Lexicon

2 floating-point-no:
<syntax> = Formula
<icon> = X

<sem> = (numerical-value X)

3 horizontal-line:
<syntax> = vert-infix-op
<icon> = X
<sem> = divide

Let us assume the input to be the icons



arranged as shown. We will note them as <5>, <h-line>, and <2>, respectively, in the trace which follows. We have to pick an order for processing these input icons. Arbitrarily, we will process the icons top to bottom. We also will be faced with the choice between Actions 1 or 2 of *main loop*. Again, arbitrarily, we'll choose Action 2 (processing items in set A) over action 1 (processing another input icon) whenever there are are items in set A to process. Lastly, the algorithm gives us the freedom of ordering the items we choose from set A. We will process each of these items in the order in which they were put into A.

The algorithm will produce states in the order shown below:

1. s_1=[2,nil,{<5>}] is added at $T_{<5>}$ through Action 1 of main loop.
2. s_2=[3,nil,{<h-line>}] is added at $T_{<5>}$ through Action 1 of main loop.
3. s_3=[1,arg_1,{<h-line>}] is added at $T_{<h\text{-}line>}$ through procedure *propose*.
4. s_4=[1,arg_2,{<h-line>,<5>}] is added at $T_{<h\text{-}line>}$ through procedure *expand*, advancing s_3 with s_1.
5. s_5=[2,nil,{<2>}] is added at $T_{<2>}$ through Action 1 of main loop.
6. s_6=[1,nil,{<h-line>,<5>,<2>}] is added at $T_{<<5><h\text{-}line><2>>}$ through procedure *complete*, advancing s_4 with s_5.
7. The procedure halts with success, s_6 satisfying the conditions.

5. RELATED WORK

We first compare related work in grammar formalisms followed by related approaches to parsing visual languages.

Of other visual grammar frameworks we are aware of, our proposal differs in the following two respects:

1. The functional role of heads and arguments. Characteristic of the linguistic roots of our approach, we assign the functional roles of head and arguments to elements in the rule body. What motivates this move? First, we assume that these syntactic roles bear a close, if not one-to-one, relationship to predicates and arguments in the semantics. In our opinion such a commitment makes it easier to coordinate incremental syntactic and semantics processing important in the parsing of visual interface languages, and it also tends to produce grammars that have more meaningful and transparent syntactic and semantic constituents. We are not aware of any such committment in competing visual grammar approaches that do discuss semantics. Second, assuming that heads of phrases tend to offer constraints on the syntactic and semantic properties of their arguments, it becomes possible to take advantage of the pruning power of these constraints through the use of head-driven parsing and generation algorithms (Kay 1989; Satta and Stock 1989; Shieber et al. 1989).

2. The domain of spatial relations. As with Helm and Marriott (1990), our formalism allows the grammar to state any number of relational constraints among any elements within the domain of a single rule. While the formalism used by Anderson (1968) differs in several other respects, he too allows spatial relations to be stated over such a domain. Unlike Golin and Reiss (1989), we do not presume that it is possible to state constraints among elements arbitrarily distant in a derivation tree. Unlike the most recent grammars of the SIL-ICON system (Crimi et al. 1989), we do not confine the expression of spatial constraints to a single relation among pairs of elements that are adjacent in a rule body. In our opinion, most visual languages in practice, complex mathematics formulae among them, need the additional expressiveness of our formalism over the latter group of proposals.

As for parsing, the algorithm we have outlined is unique among visual language parsers, as far as we know, in allowing for maximally flexible enumeration. We have motivated this design feature in the context of our goal to provide parsing tools and help facilities for interface languages, where temporal ordering of the input cannot be assumed to match systematic spatial enumeration procedures.

The other distinguishing feature of the parsing algorithm is its disassociation of the parse table from any particular set of spatial relations used by the grammar. We take this to be a strength in that the algorithm is thus extremely general, although we concede that without exploring the spatial component more fully we cannot provide a complete solution to any particular visual language domain nor can we determine the computational complexity of our algorithm. The crux of our approach is to propose a particular form of indexing of the grammar table that makes use of icons and icon sets (covers). In future work, we will explore the complexity of this algorithm when paired with sets of assumptions regarding the spatial relations assumed by the grammar.

6. CONCLUDING REMARKS

This paper concentrated on basic rule proposing and combining methods rather than on particular treatments of visual relations and representations. We expect to have more to say on these topics in future work. Other areas we expect to follow up on include the problem of nonmonotonicity inherent in allowing users to edit or alter their input, the problem of offering help to users in an incremental parsing situation, and various problems associated with reversing the grammars shown here in connection with generation of visual output from the semantics of underlying data.

Although we have been applying Relational Unification Grammars in graphical domains, there is reason to suppose that such extensions of unification grammars may prove

useful for natural languages as well. In particular, using relations such as case and gender agreement in place of left- and right-adjacency as the foundation for grammatical description may prove superior for so-called free word order languages. We expect that the parsing algorithm presented here would apply in such cases.

7. ACKNOWLEDGEMENTS

This work has been carried out under the sponsorship of the MCC Human Interface Laboratory, directed by Bill Curtis. The paper is to a large extent a revision and extension of an earlier paper coauthored with Louis Weitzman (Wittenburg and Weitzman 1990), who together with Jim Talley has worked closely with the author in developing the concepts and building the systems discussed here. Other colleagues I wish to thank include Rich Cohen for his comments on early versions of this paper and for his support of the HITS blackboard technologies used in our implementations, Chinatsu Aone for discussions on the rule formalism, and Gale Martin and Jay Pittman of the neural net character recognition group for getting me involved in this project in the first place.

REFERENCES

Aho, Alfred V., and Jeffrey D. Ullman. 1972. "The Theory of Parsing, Translation, and Compiling," Prentice Hall.

Anderson, Robert H. 1968. "Syntax-Directed Recognition of Hand-Printed Two-Dimensional Mathematics," in M. Klerer and J. Reinfelds (eds.), Interactive Systems for Experimental Applied Mathematics, Academic.

Avery, James. 1988. "Interactive Worksurface: An Interface Paradigm for Sketchable Things," MCC tech report no. ACA-HI-127-88.

Chang, Shi-Kuo. 1988. "The Design of a Visual Language Compiler," in Proceedings of the 1988 IEEE Workshop on Visual Languages, October 10-12, Pittsburgh, PA.

Crimi, C., A. Guercio, G. Tortora, and M. Tucci. 1989. "An Intelligent Iconic System to generate and to interpret Visual Languages," in Proceedings of the 1989 IEEE Workshop on Visual Languages, October 4-6 1989, Rome, Italy.

Fu, K.S. 1974. Syntactic Methods in Pattern Recognition. Academic.

Golin, Eric J., and Steven P. Reiss. 1989. "The Specification of Visual Language Syntax," in Proceedings of the 1989 IEEE Workshop on Visual Languages, October 4-6 1989, Rome, Italy.

Helm, Richard, and Kim Marriott. 1990. "Declarative Specification of Visual Languages," in Proceedings of the 1990 IEEE Workshop on Visual Languages, October 4-6, Skokie, Illinois.

Kay, Martin. 1980. "Algorithm Schemata and Data Structures in Syntactic Processing," Xerox Palo Alto Research Center, tech report number CSL-80-12.

Kay, Martin. 1989. "Head-Driven Parsing," in Proceedings of the International Workshop on Parsing Technologies, 28-31 August 1989, Pittsburgh, PA, Carnegie Mellon.

Kaplan, Ronald. 1973. "A General Syntactic Processor," in R. Rustin (ed.), Natural Language Processing, pp. 193-241, New York: Algorithmics.

Martin, Gale, James Pittman, Kent Wittenburg, Richard Cohen, and Tom Parish. 1990. Sign Here, Please: State of the Art, Computing without Keyboards. BYTE magazine, July 1990.

Satta, Georgio, and Oliviero Stock. 1989. "Head-Driven Bidirectional Parsing: A Tabular Method," in Proceedings of the International Workshop on Parsing Technologies, 28-31 August 1989, Pittsburgh, PA, Carnegie Mellon.

Shieber, Stuart. 1989. Parsing and Type Inference for Natural and Computer Languages, Technical note 460, SRI International.

Shieber, Stuart. 1986. An Introduction to Unification-Based Approaches to Grammar. Center for the Study of Language and Information, Stanford University.

Shieber, Stuart. 1985. Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms. In Proceedings of the 23rd Meeting of the Association for Computational Linguistics, 8-12 July 1985, University of Chicago.

Shieber, Stuart, Gertjan van Noord, Robert Moore, and Fernando C. N. Pereira. 1989. "A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms," in Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics, 26-29 June 1989, Vancouver.

Tomita, Masaru. 1989. "Parsing 2-Dimensional Language," in Proceedings of the International Workshop on Parsing Technologies, 28-31 August 1989, Pittsburgh, PA, Carnegie Mellon.

Wittenburg, Kent, and Louis Weitzman. 1990. "Visual Grammars and Incremental Parsing for Interface Languages," in Proceedings of the 1990 IEEE Workshop on Visual Languages, October 4-6, Skokie, Illinois.

PARSING 2-D LANGUAGES WITH POSITIONAL GRAMMARS

Gennaro Costagliola and Shi-Kuo Chang

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
gencos@speedy.cs.pitt.edu
Phone: (412) 624-8836
FAX: (412) 624-8465

ABSTRACT

In this paper we will present a way to parse two-dimensional languages using LR parsing tables. To do this we describe two-dimensional (positional) grammars as a generalization of the context-free string grammars. The main idea behind this is to allow a traditional LR parser to choose the next symbol to parse from a two-dimensional space. Cases of ambiguity are analyzed and some ways to avoid them are presented. Finally, we construct a parser for the two-dimensional arithmetic expression language and implement it by using the tool Yacc.

INTRODUCTION

One of the latest approaches in parsing 2-D languages has been presented by Tomita in [37], where he introduces a 2-D Chomsky Normal Form grammar and constructs extensions to the two-dimensional case of Earley's and LR parsing algorithms.

In this paper, we present an extension of a context-free grammar by explicitly describing the positional relations between the elements (terminals and non-terminals) in the right hand-side of each production rule of the grammar. As these relations can be very general, the resulting grammar can be seen as a generalization of Tomita's 2-D Chomsky Normal Form grammar where only horizontal and vertical relations are allowed.

The resulting parser for such a positional grammar is constructed by simply adding a column to the LR parsing table. This column contains the position of the next symbol to be shifted, for each state. Unlikely from the 2-D LR parsing algorithms given in [37], our parser slightly modifies the original LR parsing algorithm, so that the tool Yacc can be easily used to construct a two-dimensional parser for a positional grammar.

Furthermore, we analyze cases of ambiguity, give some ways to avoid them and then present a general methodology to parse two-dimensional patterns applying it to the case of the two-dimensional arithmetic expressions.

Many other approaches have been proposed till now in high dimensional syntactic pattern representation and recognition. Each of them is based on the particular data structure used for representing the pictures: a string, an array, a tree, a graph, and a plex.

One of the first approaches is given by a traditional string grammar in which more general relations (HOR, VER, ABOVE, LEFT, etc.), other than concatenation, are allowed among primitives in the pattern [2, 8, 16]. Shaw, by attaching a "head" and a "tail" to each primitive, has used four binary operators for defining binary concatenation relations between primitives. A context-free string grammar is used to generate the resulting Picture Description Language (PDL) [16, 31].

Another interesting approach using a string grammar, has been given in [5] where each primitive has associated spatial attributes.

A simple two-dimensional generalization of string grammars is to extend grammars for one-dimensional strings to two-dimensional arrays [23, 28, 35, 38]. The primitives are the array elements and the relation between primitives is the two-dimensional concatenation.

Pfalz and Rosenberg have extended the concept of string grammar to grammars for labeled graphs called webs [16, 17, 26, 27, 29]. These grammars were originally suggested as a syntactical formalism for data structure useful in image analysis. An application of graph languages for describing scenes is of frequent occurrence in the literature dealing with image processing, whereas the use of graph grammars for pattern recognition is rare (for this purpose tree grammars are applied instead [3, 17, 18, 22, 30, 32]). Difficulties concerning building a syntax analyzer for graph grammars are causes of this situation. Recently, however, parsing methods for a particular kind of graph grammar have been proposed, and an efficient parsing, close to the parsing efficiency of tree languages, has been obtained [15, 21, 33].

Based on an idea in the work of Narasimhan [24], Feder [14] has formalized a "plex" grammar which generates languages with terminals having an arbitrary number of attaching points in order to connect to other primitives or subpatterns. The primitives of the plex grammar are called N-Attaching Point Entities (NAPEs). Plex structures defined by a plex grammar may be viewed as a hypergraph, with each NAPE corresponding to a hyperedge. Therefore this kind of plex grammar is a more general model than that of graph gram-

mar. Until recently, however, very little was known about the parsing method for plex grammars. Recently, a parsing method has been developed [25] to achieve more efficient parsing of plex grammars, by adapting Earley parsing algorithm, [13].

The paper is organized as follows. In Section 2 the positional grammar is defined, and some examples are given. In Section 3 the extension of the LR parser, named positional LR (pLR) parser, is presented along with a description of the pLR parsing tables and of the parsing algorithm. In Section 4 considerations of ambiguity are given along with the construction of a pLR parser for the arithmetic expression grammar. In Section 5 we present the general methodology for parsing 2-D languages generated by a positional grammar. The conclusions are in Section 6.

POSITIONAL GRAMMARS

The parser we are going to present recognizes pictorial languages generated by *positional* grammars.

Definition 2.1

A context-free *positional grammar* PG can be represented by 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,
$N \cap T = \emptyset$,
$S \in N$ is the *starting* symbol,
P is a finite set of *productions*
POS is a finite set of *positional relation* identifiers
$POS \cap (N \cup T) = \emptyset$,
PE is an *evaluation rule*

Each production in P has the following form:

$$A \rightarrow \alpha_1 REL_1 \alpha_2 REL_2 \cdots REL_{m-1} \alpha_m \qquad m \geq 1$$

where $A \in N$, each α_i is in $N \cup T$ and each REL_i is in POS. |

Each positional relation REL_i gives information about the relative position of α_{i+1} with respect to α_i. In the following, the words "positional grammar" will always refer to a context-free positional grammar.

While in a string grammar the only possible positional relation is the string concatenation, in a positional grammar other positional relations can be defined and then used for describing high dimensional languages. When parsing, this positional information will be useful for letting the scanner know where the next symbol to parse is.

Some simple examples of positional relations on a Cartesian plane:

String concatenation or adjacent horizontal concatenation

$AHOR = \{(p_1, p_2) : p_1$ and p_2 are pictures horizontally concatenated with alignment of their centroids$\}$

Adjacent vertical concatenation

$AVER = \{(p_1, p_2) : p_1$ and p_2 are pictures vertically concatenated with alignment of their centroids$\}$

Upper horizontal concatenation

$UHOR = \{(p_1, p_2) : p_1$ and p_2 are pictures horizontally concatenated with alignment of the centroid of p_1 and the up-most element of $p_2\}$

Horizontal concatenation

$HOR = \{(p_1, p_2) : p_1$ and p_2 are pictures and location$(p_1) = (x, y)$ and location'$(p_2) = (x', y')$ and the position (x', y') is *feasible* and $x' > x\}$

Vertical concatenation

$VER = \{(p_1, p_2) : p_1$ and p_2 are pictures and location$(p_1) = (x, y)$ and location'$(p_2) = (x', y')$ and the position (x', y') is *feasible* and $y' < y$ and $x' \leq x\}$

where a picture is a spatial arrangement of one or more symbols, location(p) is a function returning the position of a symbol of the picture p and a feasible location is a location that has not been made unfeasible by another symbol or by the side effect of an evaluation rule, as it will be seen in the following.

Definition 2.2

An *evaluation rule* PE is a function whose input is a string

$$p_1 REL_1 p_2 REL_2 \cdots REL_{m-1} p_m \qquad m \geq 1$$

where each p_i is a picture and each REL_i is a positional relation; its output is a picture whose elements $p_1, p_2, \ldots, p_m$ are disposed in the space such that

$$(p_i, p_{i+1}) \in REL_i \qquad 1 \leq i \leq m-1.$$

The evaluation of the positional relations is meant to be sequential from left to right. As side effects can be generated for any evaluation, an *evaluation rule* is *simple* if no side effects are involved. |

A possible side effect of the evaluation of a relation is to make certain positions in the space unfeasible. As the evaluation is sequential, each evaluation inherites the side effects generated by the previous evaluations.

Some examples of applications of the simple evaluation rule follow:

PE("a . b . c . d") = $a\ b\ c\ d$

PE("a *VER* b *HOR* c") =
$$\begin{matrix} & a & \\ b & & c \end{matrix}$$

PE("a *AVER* b") =
$$\begin{matrix} a \\ b \end{matrix}$$

where the positional relations '.', *VER*, *HOR* and *AVER* are defined as above.

The following definitions are understood to be with respect to a particular positional grammar G.

We write $\Pi \Rightarrow \Sigma$ if there exist Δ, Γ, A, η such that $\Pi = \Gamma A \Delta$, $A \rightarrow \eta$ is a production and $\Sigma = \Gamma \eta \Delta$.

We write $\Pi \Rightarrow^* \Sigma$ (Σ is derived from Π) if there exist strings $\Pi_0, \Pi_1 \cdots \Pi_m$ ($m \geq 0$) such that

$$\Pi = \Pi_0 \Rightarrow \Pi_1 \Rightarrow \cdots \Rightarrow \Pi_m = \Sigma$$

The sequence $\Pi_0, ..., \Pi_m$ is called a derivation of Σ from Π. A *positional sentential form* is a string Π such that $S \Rightarrow^* \Pi$. A *positional sentence* is a positional sentential form with only terminal symbols. A *pictorial form* is the evaluation of a positional sentential form. A *picture* is a pictorial form with only terminal symbols. The *pictorial language defined by a positional grammar L(G)* is the set of its pictures.

Some examples of positional grammars:

2.1) The following grammar generates the strings of the form $a \cdots ab \cdots b$ with equal number of a's and b's.

N = {S}
T = {a, b}
POS= {.}
PE is the simple evaluation rule
P = {
S := a . S . b | a . b
}

The positional operator . is defined as above. A positional sentence of this grammar is: a . a . a . b . b . b and the corresponding picture is: aaabbb.

This example shows that every context-free string language can be represented by a positional grammar.

2.2) The following grammar generates an upper-right corner with variable length of the edges.

N = {Corner, HLine, VLine}
T = {*dot*}
S = Corner
POS= {*UHOR*, *AHOR*, *AVER*}
PE is the simple evaluation rule
P = {
Corner := HLine *UHOR* VLine
HLine := HLine *AHOR dot* | *dot*
VLine := VLine *AVER dot* | *dot*
}

where *UHOR*, *AHOR* and *AVER* are defined as above. A positional sentence of this grammar is:

dot AHOR dot AHOR dot AHOR dot UHOR dot AVER dot AVER dot AVER dot

Replacing *dot* with the character '.', the corresponding picture is:

2.3) The following grammar generates two-dimensional arithmetic expressions using the binary operations addition and division:

N = {E, T, F}
S = E
T = {+, *hbar*, (,), *id*}
POS= {*HOR*, *VER*}
PE is the evaluation rule defined below
P = {
E := E *HOR* + *HOR* T | T
T := T *VER hbar VER* F | F
F := (*HOR* E *HOR*) | *id*
}

The evaluation rule is so defined (see Figure 2.1):

PE(p_1 *HOR* p_2):
The evaluation of *HOR* will give coordinates (x, y) to location(p_1) and (x', y') to location(p_2) such that (p_1, p_2) $\in$ *HOR*. Moreover it will make unfeasible each position belonging to any of the following sets:
$\{(x, y_1) : y \leq y_1 \leq m\}$
$\{(x_1, y_2) : x < x_1 < x' \text{ and } 0 \leq y_2 \leq m\}$
$\{(x', y_3) : y' \leq y_3 \leq m\}$
where $m \geq 1$ is an upper bound on the y-coordinate in the two-dimensional space.

PE(p_1 *VER* p_2):
The evaluation of *VER* will give coordinates (x, y) to location(p_1) and (x', y') to location(p_2) such that (p_1, p_2) $\in$ *VER*. Moreover it will make unfeasible each position belonging to any of the following sets:
$\{(x_1, y) : 0 \leq x_1 \leq x\}$
$\{(x_2, y_1) : 0 \leq x_2 \leq x \text{ and } y' < y_1 < y\}$
$\{(x_3, y') : 0 \leq x_3 \leq x'\}$

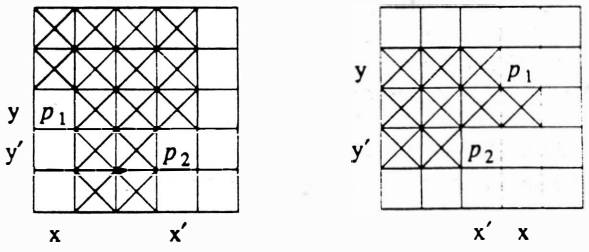


Figure 2.1. {p_1 *HOR* p_2} and {p_1 *VER* p_2}

A positional sentence of this grammar is:

id HOR + HOR (HOR id HOR + HOR id HOR) VER hbar VER id HOR + HOR id

Replacing *hbar* with an horizontal bar, according to the definitions of *HOR*, *VER* and PE, there are many possible pictures corresponding to the evaluation of this positional sentence, but all of them can be mapped into the following one:

$$id + \frac{(id + id)}{id} + id$$

that is still a picture of this language.

POSITIONAL LR PARSERS

Positional LR parsers (pLR parsers) are nothing else but a generalization of the LR parsers. The model of a pLR parser is given by:

1) Input
2) Positional operators
3) pLR Parsing Table
4) pLR Parsing Program
5) Stack
6) Output

as shown in Figure 3.1.

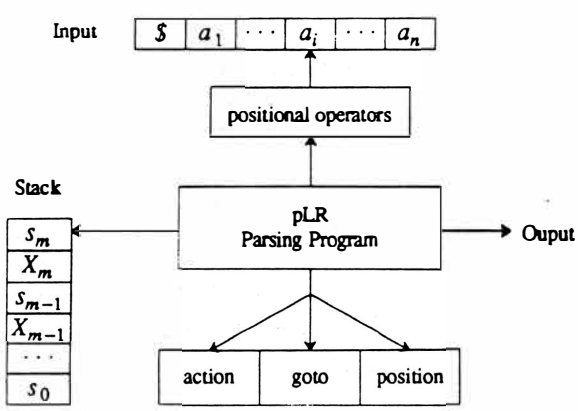


Figure 3.1. The model of a pLR Parser

The input

The input to a pLR parser is a spatial arrangement of tokens, or, in other words, a symbolic picture where each symbol is a token. Such an input is represented by an array w (the input tape) where each token is stored, a list Q of couples (*pos*, *i*) where *pos* is the spatial position of the token w[*i*], and a starting index that points to the first token to parse. The association between a position and a token allows us to reach a token in w each time its spatial position has been given and viceversa. The input tape is, then, no longer required to be accessed sequentially but rather, according to the positional requirements given by the parser.

In this context, the definition of the sequential end-of-string marker must be extented. In fact, the end-of-string marker hides an operational aspect: when parsed, it signals that no symbols to parse are left. While in a sequential scanning nothing must be done other than recognizing the '$' character, in a non-sequential scanning such operational aspect must be made explicit. Before returning an end-of-input symbol, the scanner has to check whether all the symbols have been parsed.

In a pLR parser, the end-of-input marking is implemented by storing the symbol '$' in location 0 of the input tape, and defining the *end-of-input operator* ANY as a function whose return value is 0 if all the symbols in the input tape have been parsed and 'error' otherwise.

The positional operators

For each positional relation we define a positional operator with the same name. Such an operator is a function that takes in input the index in the tape of the last token parsed, calculates a new position and then returns the index of the next token to parse, by consulting the list Q.

Definition 3.1

Given a positional grammar PG = (N, T, S, P, POS, PE) and a relation *REL* ∈ POS, then for all α, β ∈ N ∪ T such that "α *REL* β" occurs on the right hand-side of a production rule in PG, the *corresponding positional operator* REL is defined as follows:

REL(i) = j iff i is the index in w of 'a', the last token parsed to reduce α, and j is the index in w of 'b', the first token to parse to reduce β. |

Examples:

3.1) In the grammar of Example 2.2, the corresponding operators for POS can be defined as follows:

UHOR(i) = AHOR(i) = j iff location(w[i]) = (x, y) and location(w[j]) = (x+δ, y).

AVER(i) = j iff location(w[i]) = (x, y) and location(w[j]) = (x, y-δ).
where δ is the distance between each couple of dots.

3.2) For the arithmetic expression grammar the operators HOR and VER can be defined as follow:

HOR(i) = j iff location(w[j]) is the highest spatial position in the first non-empty column on the right of location(w[i]).

VER(i) = j iff location(w[j]) is the spatial position on the left of location(w[i]) such that it is the leftmost position in the first non-empty row below location(w[i]).

The Positional LR Parsing Table

Besides the "action" and "goto" columns of an LR parsing table, the pLR parsing table contains an additional column called "position". The positional operators SP, ANY and the names of the positional operators are the elements of this new column. SP returns the starting index given in input with the picture and ANY is the operator defined above. All the names

in the column "position" can be considered as pointers to the code implementing the operators.

As the construction of the "position" column does not affect the other entries of the original LR parsing table, we can use the traditional three techniques (with some variations) for having Simple pLR, canonical pLR and LookAhead pLR parsers.

A pLR(0) item of a positional grammar PG is a production of PG with a dot at some position of the right side. A dot, however, can never be between a positional operator identifier and either a terminal or a non terminal, in this order. Thus, a production A → SP X REL_1 Y REL_2 Z yields the four items:

A → .SP X REL_1 Y REL_2 Z
A → SP X .REL_1 Y REL_2 Z
A → SP X REL_1 Y .REL_2 Z
A → SP X REL_1 Y REL_2 Z .

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the first item above indicates that we hope next to see a pattern derivable from XYZ starting from position SP. The second item indicates that we have just seen on the input a pattern derivable from X and that we hope next to see a pattern derivable from YZ starting from the position specified by the operator associated to REL_1.

If PG is a grammar with starting symbol S, then PG', the augmented positional grammar for PG, is PG with a new starting symbol S' and production S' := SP S.

Example 3.3

Let us consider the following positional grammar generating an horizontal concatenation of a block of squares, an arrow and another block of squares

(1) S := B1 *HOR* => *HOR* B2
(2) B1 := C *HOR* C
(3) C := square *VER* square
(4) B2 := R *VER* R
(5) R := square *HOR* square

Here the definition of PE is as in Example 2.3. The canonical collection of sets of pLR(0) items for this grammar follows next, along with the position values. The goto function for this set of items is shown as the transition diagram of a deterministic finite automaton in Figure 3.2 and the resulting Positional LR parsing table is given in Figure 3.3.

I_0: S' := .SP S — position[0] = {SP}
S := .B1 HOR => HOR B2
B1 := .C HOR C
C := .square VER square

I_1: S' := SP S. — position[1] = {ANY}

I_2: S := B1 .HOR => HOR B2 — position[2] = {HOR}

I_3: B1 := C .HOR C — position[3] = {HOR}
C := .square VER square

I_4: C := square .VER square — position[4] = {VER}

I_5: S := B1 HOR => .HOR B2 — position[5] = {HOR}
B2 := .R VER R
R := .square HOR square

I_6: B1 := C HOR C . — position[6] = {HOR}

I_7: C := square VER square . — position[7] = {HOR}

I_8: S := B1 HOR => HOR B2 . — position[8] = {ANY}

I_9: B2 := R .VER R — position[9] = {VER}
R := .square HOR square

I_{10}: R := square .HOR square — position[10] = {HOR}

I_{11}: R := square HOR square . — position[11] = {VER,ANY}

I_{12}: B2 := R VER R . — position[12] = {ANY}

Note that in the construction of each closure, the positional operators HOR and VER are ignored by the dot. This information is instead caught by the position array.

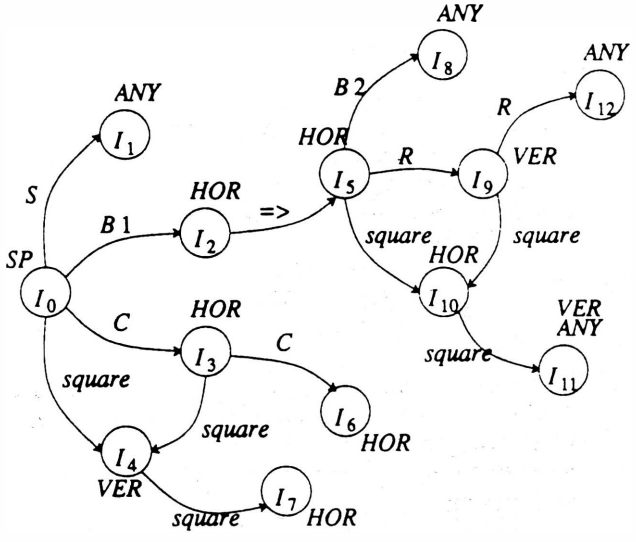


Figure 3.2. Transition diagram

state	action			goto					position
	square	=>	$	S	B1	B2	C	R	
0	s4			1	2		3		SP
1			acc						ANY
2		s5							HOR
3	s4						6		HOR
4	s7								VER
5	s10					8		9	HOR
6		r2							HOR
7	r3	r3							HOR
8			r1						ANY
9	s10							12	VER
10	s11								HOR
11	r5		r5						ANY VER
12			r4						ANY

Figure 3.3. A Simple pLR parsing table

Details on the algorithm for the construction of a Positional LR parsing table can be found in [9, 10].

The Positional LR Parsing Algorithm

The pLR algorithm is a simple extension of Algorithm 4.7 in [1]; the only differences regard the form of the input and the setting of the pointer to the next symbol.

The input is now given by a picture p represented by an array of tokens w, a starting index in w, and a list Q of couples (*pos*, *i*); the specification of a set of positional operators, and the pLR parsing table with functions "action", "goto" and "position" for a positional grammar PG.

Each time the pLr parser reaches a state in the recognition of the pattern, the next symbol to parse is determined by using the positional operator associated to that state. As in LR parsing, a same symbol cannot be considered more than once.

Details on the Positional LR parsing algorithm can be found in [9, 10].

Examples

3.4) Figure 3.4 shows the parsing action, goto and position of a canonical pLR parsing table for the following linear positional grammar for the vertical concatenation of two strings both of the type "c · · · cd".

(1) S := C *VER* C
(2) C := c *AHOR* C
(3) C := d

where the evaluation rule is simple when applyed to *AHOR* and defined as in Example 2.3 when applied to *VER*. Using the parsing table in Figure 3.4 and applying the pLR parsing algorithm, it can be verified that the following picture

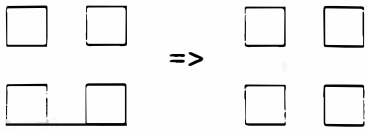
ccccccccccd
ccccd

is in the described language.

state	action			goto		position
	c	d	$	S	C	
0	s3	s4		1	2	SP
1			acc			ANY
2	s6	s7			5	VER
3	s3	s4			8	AHOR
4	r3	r3				VER
5			r1			ANY
6	s6	s7			9	AHOR
7			r3			ANY
8	r2	r2				VER
9			r2			ANY

Figure 3.4. A canonical pLR parsing table

3.5) Given the grammar in Example 3.3, using the parsing table in Figure 3.3 and applying the pLR parsing algorithm, it can be verified that the following picture



is accepted. In particular, note that the parser drives the scanning of the input such that the first block is visited by columns, and the second block by rows, according to the productions of the grammar. All the other ways of scanning this input are not taken into consideration.

AMBIGUITY CONSIDERATIONS

In this Section we will show that conflicts in positions can lead to conflicts in the "action" part of the parsing table even if it has no multiple entries.

In Section 2 we gave a two-dimensional version of the grammar given in [1] for arithmetic expressions. We will show now that this grammar is not pLR(1) from the fact that it has conflicts regarding the position of the next symbol. Let us consider the following pictorial form:

$$\frac{T}{id} + id$$

assuming that T has already been reduced.

After reducing T, the parser has to decide whether to choose '*hbar*' in vertical reading, or '+' in horizontal reading. Both the alternatives are valid: if '*hbar*' is chosen, then the parser has to shift, otherwise it has to reduce. One possibility for avoiding this conflict is to assign priority to each positional operator. In this example we could decide that the vertical reading has always higher priority than the horizontal one. This would respect the priority between '*hbar*' and '+' implicitly given in the grammar. But, if this other example is considered

$$\frac{(T + id)}{id} + id$$

the priority resolution will fail. In fact, in this case, after reading T, we want to move horizontally because of the parenthesis, and not vertically.

Another possibility for avoiding this conflict is to give a "smart" representation of the two-dimensional pattern deriving it from techniques of image analysis like dominancy [4, 12]. Last but not least, we can construct an equivalent pLR(1) grammar as it is normally done for solving conflicts in LR parsers. Following these ideas, the pLR(1) grammar for the arithmetic expressions has been constructed:

(0) E' := SP E
(1) E := E *HOR* + *HOR* T

(2) E := T
(3) T := T' *VER* F
(4) T := F
(5) F := (*HOR* E *HOR*)
(6) F := *id*
(7) T' := T' *VER* F'
(8) T' := F'
(9) F' := ⌊ *HOR* E *HOR* ⌋
(10) F' := *id*

Figure 4.1 shows the resulting pLR(1) parsing table for this grammar.

Note that the terminals *id*, (, and) have been duplicated as well as the non-terminals T and F. Moreover, rules (3), (4), (5) and (6) have been duplicated in rules (7), (8), (9) and (10). The new grammar, then, has a particular section dedicated to the generation of the numerator of any division.

During the recognition, this allows us to decide whether the expression to be parsed is the numerator of a division or not. In particular, the new terminals ⌊ and ⌋ mark the beginning and the end of any complex numerator, respectively, and the terminal *id* is the simple numerator.

state	action							goto						position
	$	*id*	+	)	(	*id*	⌋	⌊	E'	E	T	F	T'	F'	
0		s7			s5	s6		s8		1	2	4	3	9	SP
1	acc		s10												(HOR (ANY
2	r2		r2	r2			r2								HOR
3		s7			s5	s6		s8				12		13	VER
4	r4		r4	r4			r4								HOR
5		s7			s5	s6		s8		14	2	4	3	9	HOR
6	r6		r6	r6			r6								HOR
7		r10			r10	r10		r10							VER
8		s7			s5	s6		s8		15	2	4	3		HOR
9		r8			r8	r8		r8							VER
10		s7			s5	s6		s8			11	4	3	9	HOR
11	r1		r1	r1			r1								HOR
12	r3		r3	r3			r3								HOR
13			r7	r7			r7								HOR
14			s10	s16											HOR
15			s10				s17								HOR
16	r5		r5	r5			r5								HOR
17		r9			r9	r9		r9							VER

Figure 4.1. pLR parsing table for arithmetic expressions

A trace for the acceptance of the following patterns can be easily constructed

$$\begin{matrix} \lfloor id + id \rfloor \\ id \end{matrix} + id \qquad \begin{matrix} id \\ id + id \\ id \end{matrix}$$

AN IMPLEMENTATION

The general methodology to parse pLR languages is the following:

I. Define a general data structure to represent the two-dimensional symbolic pictures.

II. Define the positional relations and operators meant to relate objects in the patterns, and construct the pLR positional grammar, if possible, to describe the language.

III. Convert the general data structure into the input to the parser as defined in Section 3.

IV. Construct the parser.

Point I requires a general data structure to represent the original symbolic picture input. This can be a matrix of symbols, or an iconic index, i. e., an analogous linear representation based on the projections of the symbols: the 2-D string as defined in [6], or, for high dimensional symbolic patterns, the Gen_string, [11]. As the whole parsing model presented is extensible to the n-D case (n >2) just considering positional relations and operators for the n-dimensional space, we will make use of the Gen_string iconic index. The characteristics of it and the algorithms to derive it from a high dimensional pattern are given in [11]. In the proposed implementation, each element of the Gen_string is a token. A lexical analyzer to construct such a Gen_string can be obtained by using the same actions described above, but allowing the elements of the general data structure (another Gen_string) to be elementary items or pixels.

Point II requires the construction of the pLR linear positional grammar along with the positional operators.

Point III requires routines for the conversion of the general data structure into an array of tokens w, a starting index in w, SP, and an association list Q of positions and tokens. In particular the list Q must be implemented such that the positional operators can be executed efficiently.

Finally, Point IV requires the construction of the parser. As a result of Theorem 7.1 in[9], this can be done by translating the positional LR grammar into an LR grammar with actions and then by using the tool Yacc, [20].

As an example of the construction given in that Theorem, let us consider the the positional LR grammar for the arithmetic expressions. The resulting LR context free grammar with actions is:

(1) E := E + {HOR()} T
(2) E := T
(3) T := T' F
(4) T := F
(5) F := ({HOR()} E) {HOR()}
(6) F := *id* {HOR()}
(7) *T'* := *T' F'*
(8) *T'* := *F'*
(9) *F'* := ⌊ {HOR()} E ⌋ {VER()}
(10) *F'* := *id* {VER()}

An implementation by Yacc for this grammar, using the Gen_string representation, has been developed at the Department of Computer Science of the University of Pittsburgh. The implementation consists of the following:

The function *get_gs()*: the Gen_string representing a two-dimensional arithmetic expression is stored in a global data structure "gs". The Gen_string can be taken from a database or derived from the original pattern.

The function *gs_ir()*: the Gen_string is converted into an internal representation (data structure "spg", and others).

The functions *read_hor()* and *read_ver()*: the spatial operators HOR and VER are implemented, respectively.

The *yacc specifications* for the grammar: the functions read_hor() and read_ver() are inserted in the rules as actions. Both of them update a global variable "current" used by the function *yylex()* to select the next token to be parsed.

In the following, the results of the execution of such specifications are given. Note that the array "spg" represents the set of tokens occurring in the expression while the values of "current" give the order in which the tokens are parsed. For each token spg[i], the (x, y) coordinates are also given (the list Q). In this implementation x represents the column index in left-right progression, and y the row index in top-down progression.

Case 1

get_gs1: the input Gen_string is equivalent to

$$\frac{(99 + 501)}{6} * \frac{10}{2}$$

spg[0] = "\0"		
spg[1] = "("	x = 1	y = 1
spg[2] = "99"	x = 2	y = 1
spg[3] = "+"	x = 3	y = 1
spg[4] = "6"	x = 3	y = 2
spg[5] = "501"	x = 4	y = 1
spg[6] = ")"	x = 5	y = 1
spg[7] = "*"	x = 6	y = 2
spg[8] = "10"	x = 7	y = 1
spg[9] = "2"	x = 7	y = 2

current = 1 2 3 5 6 4 7 8 9 0 ... the result is -> 500

Case 2

get_gs2: the input Gen_string is equivalent to

$$\frac{8}{\left(\frac{4}{2}\right)} - 2$$

spg[0] = "\0"		
spg[1] = "("	x = 1	y = 2
spg[2] = "8"	x = 2	y = 1
spg[3] = "4"	x = 2	y = 2
spg[4] = "2"	x = 2	y = 3
spg[5] = ")"	x = 3	y = 2
spg[6] = "-"	x = 4	y = 2
spg[7] = "2"	x = 5	y = 2

current = 2 1 3 4 5 6 7 0 ... the result is -> 2

CONCLUSIONS

In this paper we constructed a parser for a subclass of symbolic pictorial languages. We showed that this class contains the context-free string languages and that a complex language like the two-dimensional arithmetic expression language can be parsed by the proposed model. We also showed that this class has a real nice property: the possibility to be parsed in a very simple way by using an existing tool.

At the moment we are investigating the extension of universal parsers like Earley's ([13]) and Tomita's ([36]) algorithms by applying the same technique used for extending the LR parser. Moreover we are considering applications of the model proposed to graphics and to visual languages ([7, 12, 19, 34]).

In the future we intend to extend the subclass of pictorial languages parseable by constructing more powerful parsers. A first approach regards the extension of the concept of symbol to an N-Attaching Point Entity as defined in [14]. A second approach regards instead the possibility to have more than one positional relation between two symbols. In this way a symbol can be connected to non-adjacent symbols, too.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques, and tools*, Addison Wesley, 1985.

[2] H.G. Barrow and J.R. Popplestone, "Relational Descriptions in picture processing," *Machine Intelligence*, vol. 6, pp. 377-396, 1971.

[3] N.S. Chang and K.S. Fu, "Parallel Parsing of Tree Languages for Syntactic Pattern Recognition," *Pattern Recognition*, vol. 11, no. 3, pp. 213-222, 1979.

[4] S.-K. Chang, "A Method for the Structural Analysis of Two Dimensional Mathematical Expressions," *Information Sciences*, vol. 2, pp. 253-272, 1970.

[5] S.-K. Chang, "Picture Processing Grammar and its Applications," *Information Sciences*, vol. 3, pp. 121-148, 1971.

[6] S.-K. Chang, Q.Y. Shi, and C.W. Yan, "Iconic Indexing by 2-D strings," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 4, pp. 475-484, July 1984.

[7] S.-K. Chang, M.J. Tauber, B. Yu, and J.S. Yu, "A Visual Language Compiler," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 506-525, 1989.

[8] M.B. Clowes, "Pictorial Relationships - A Syntactic Approach," *Machine Intelligence*, vol. 4, Amer. Elsevier, New York, 1969.

[9] G. Costagliola and S.-K. Chang, "Parsing Linear Pictorial Languages by Syntax-Directed Scanning," *submitted to JACM*.

[10] G. Costagliola and S.-K. Chang, "DR PARSERS: a generalization of LR parsers," *Proc. of 1990 IEEE Workshop on Visual Languages*, pp. 174-180, Skokie, Illinois, USA, October 4-6.

[11] G. Costagliola, G. Tortora, and T. Arndt, "A Unifying Approach to Iconic Indexing for 2-D and 3-D Scenes," *to appear in IEEE Transactions on Knowledge and Data Engineering*.

[12] C. Crimi, A. Guercio, G. Pacini, G. Tortora, and M. Tucci, "Automating Visual Language Generation," *IEEE Transactions on Software Engineering*, vol. 16 , no. 10, pp. 1122-1135, October 1990.

[13] J. Earley, "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM*, vol. 13, pp. 94-102, 1970.

[14] J. Feder, "Plex Languages," *Information Sciences*, vol. 3, pp. 225-241, 1971.

[15] M. Flasinski, "Characteristics of edNLC-Graph Grammar for Syntactic Pattern Recognition," *Computer Vision Graphics and Image Processing*, vol. 47, pp. 1-21, 1989.

[16] K.S. Fu, *Syntactic Methods in Pattern Recognition*, Academic Press, New York and London, 1974.

[17] K.S. Fu, *Syntactic Pattern Recognition and Applications*, Prentice Hall, Inc. Englewood Cliffs, N.J., 1982.

[18] K.S. Fu and B.K. Bhargava, "Tree Systems for Syntactic Pattern Recognition," *IEEE Trans. Comput.*, vol. C-22 (12), pp. 1089-1099, 1973.

[19] E.J. Golin and S.P. Reiss, "The Specification of Visual Language Syntax," *Proc. of 1989 IEEE Workshop on Visual Languages*, pp. 105-110, Rome/Italy, October 4-6.

[20] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler," *tech. rep.*, Bell Laboratories, 1974.

[21] C.Y. Li, T. Kawashima, T. Yamamoto, and Y. Aoki, "Attribute Expansive Graph Grammar for Pattern Description and its Problem-reduction Based Processing," *Trans. IEICE*, vol. E-71 (4), pp. 431-440, Japan, 1988.

[22] S.Y. Lu and K.S. Fu, "Error-correcting Tree Automata for Syntactic Pattern Recognition," *IEEE Trans. Comput.*, vol. C-27, pp. 1040-1053, 1978.

[23] D.L. Milgram and A. Rosenfeld, "Array Automata and Array Grammars," *Information Processing* , vol. 71, pp. 69-74, North-Holland Publ., Amsterdam, 1972.

[24] R. Narasimhan, "Syntax-directed Interpretation of Classes of Pictures," *Comm. ACM*, vol. 9, pp. 166-173, 1966.

[25] K. J. Peng, T. Yamamoto, and Y. Aoki, "A New Parsing Scheme for Plex Grammars," *Pattern Recognition*, vol. 23 , no. 3/4, pp. 393-402, 1990.

[26] J. L. Pfaltz, "Web Grammars and Picture Description," *Comput. Graphics Image Processing*, vol. 1, pp. 193-220, 1972.

[27] J. L. Pfaltz and A. Rosenfeld, "Web Grammars," *Proc. of First Int. Joint Conf. Artif. Intell.*, pp. 609-619, Washington, DC, May 1969.

[28] A. Rosenfeld, *Picture Languages: Formal Models for Picture Recognition*, Academic Press, New York, San Francisco and London, 1979.

[29] A. Rosenfeld and D. L. Milgram, "Web Automata and Web Grammars," *Machine Intelligence*, vol. 7, pp. 307-324, 1972.

[30] W.C. Rounds, "Context Free Grammars on Trees," *Proc. of 10th Symp. Switching and Automata Theory*, p. 143, 1969.

[31] A.C. Shaw, "A Formal Picture Description Scheme as a Basic for Picture Processing Systems," *Information and Control*, vol. 14, pp. 9-52, 1969.

[32] Q.Y. Shi and K.S. Fu, "Efficient and Error-correcting Parsing of (attributed and stochastic) Tree Grammars," *Information Sciences*, vol. 26, pp. 159-188, 1982.

[33] Q. Y. Shi and K.S. Fu, "Parsing and Translation of Attributed Expansive Graph Languages for Scene Analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, pp. 472-485, 1983.

[34] N.C. Shu, *Visual Programming*, Van Nostrand Reinhold Company, 1988.

[35] G. Siromoney, R. Siromoney, and K. Krithivasan, "Array Grammars and Kolam," *Comput. Graphics and Image Processing*, vol. 3, pp. 63-82, 1974.

[36] M. Tomita, *Efficient Parsing for Natural Languages*, Kluwer Academic Publishers, Boston, MA, 1985.

[37] M. Tomita, "Parsing 2-Dimensional Languages," *Proceedings of the International Workshop on Parsing Technologies*, pp. 414-424, Pittsburgh, PA. Carnegie Mellon, 28-31 August 1989.

[38] P.S.P. Wang, "Recognition of Two-Dimensional Patterns," *Proc. Assoc. Comput. Mach. Nat. Conf.*, pp. 484-489, 1977.

NOTES:


The Special Interest Group on Parsing of the Association for Computational Linguistics
SIGPARSE
Sponsored by