A New Transformation into Deterministically Parsable Form for Natural Language Grammars

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

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

Abstract

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

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

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

1 Introduction

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

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

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

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

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

2 State of the Art

2.1 Marcus Parsers

2.1.1 PARSIFAL

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

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

2.1.2 Problems with the Marcus approach

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

2.1.3 Other Marcus Parsers

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

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

2.2 LR Parsers

2.2.1 Outline

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

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

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

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

2.2.2 Problems with this approach

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

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

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

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

The horse raced past the barn fell.

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

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

3 A new approach

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

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

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

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

4 Notation

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

4.1 Trees

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

Fig. 1: An *and* node.

Fig. 2: An *or* node.

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

4.2 Message passing

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

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

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

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

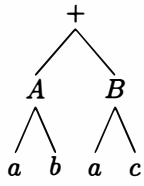


Fig. 3: Tree for grammar G_1.

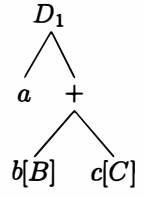


Fig. 4: Transformed version of G_1.

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

4.3 Gated *or* nodes

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

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

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

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

Fig. 5: An example grammar G_2.

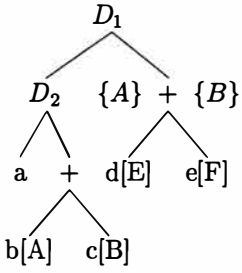


Fig. 6: Transformed version of grammar G_2

4.4 Special gated *or* nodes

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

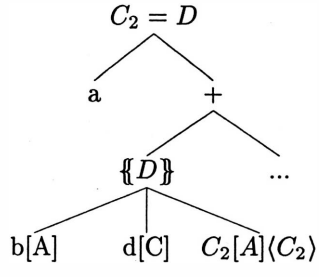


Fig. 7: Grammar G_3

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

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

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

5 Transformation Method

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

5.1 Pre-Transformations

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

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

5.2 Main Transformation

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

5.2.1 Comparison between two graph segments:

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

5.2.2 Make the unified tree:

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

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

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

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

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

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

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

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

5.3 Post-transformations

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

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

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

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

5.4 Example

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

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

Fig.8: Grammar G_3

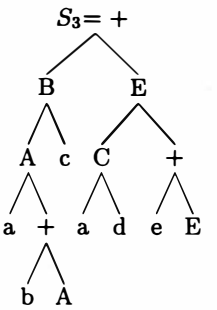


Fig.9: Start grammar G_3.

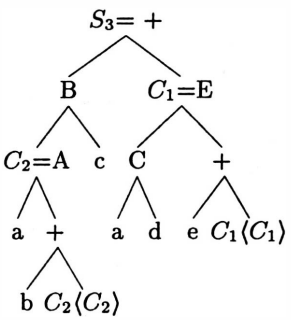


Fig.10: Mark cycles.

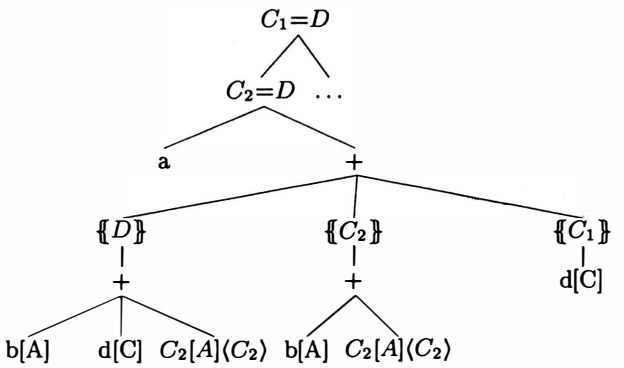


Fig.11 : Unify: build dummy chain and special gated nodes.

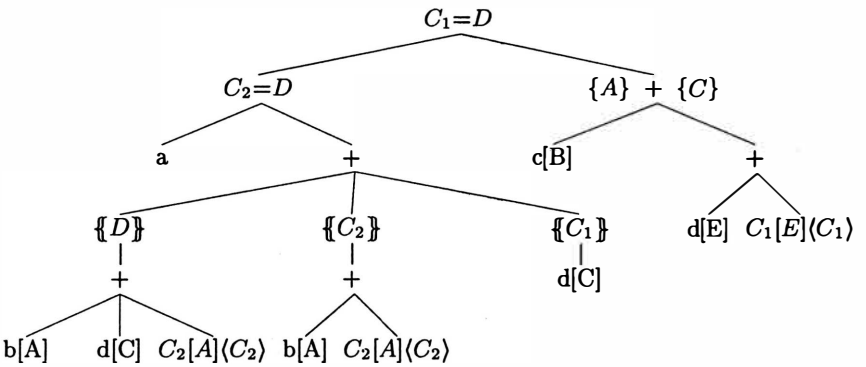


Fig.12: Unify: build gated *or* nodes.

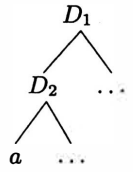
5.5 Parsing method

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

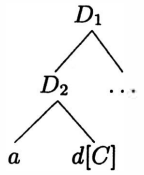
- The parser matches a node which contains messages. These messages are then passed back up the parse tree to replace the dummy nodes encountered whilst building the tree.
- The parser encounters a dummy node which has an empty node as its right child. The parser then replaces the dummy node and right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

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

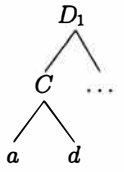
Input: $|adade$
Action: Match symbol a, build chain of dummy nodes.



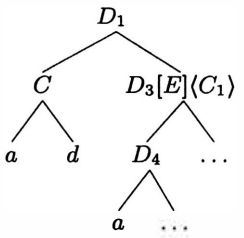
Input: $a|dade$
Action: Cycle name is unresolved, so follow the path below the dummy gate $\{\!\{D\}\!\}$ (sibling of a). Match the symbol d.



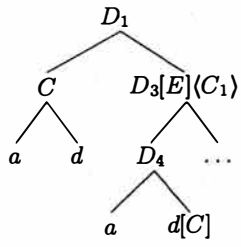
Input: $ad|ade$
Action: Pass message C back up to replace the dummy node D_2.



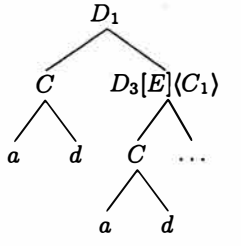
Input: $ad|ade$
Action: Match the gated node $\{C\}$ as its sibling node is now C. No symbol e, so follow the cycle C_1 with message E and cycle name resolved to C_1. Now match the symbol a.



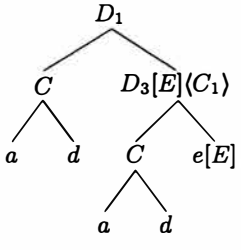
Input: $ada|de$
Action: The cycle name has been resolved to C_1, so follow the path below the special gated node $\{\!\{C_1\}\!\}$. Match the only possibility of d with message C.



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

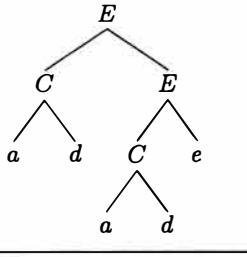


Input: *adad|e*
Action: follow the path below the gated node $\{C\}$ as its sibling node is C. Match the input symbol e with message E.



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

Parse is now finished.



6 Improvements

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

7 Conclusion

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

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

Acknowledgements

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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