L* Parsing: A General Framework for Syntactic Analysis of Natural Language

Eric K. Jones and **Linton M. Miller**

Department of Computer Science
Victoria University of Wellington
P.O. Box 600, Wellington, New Zealand
Eric.Jones@comp.vuw.ac.nz Linton.Miller@comp.vuw.ac.nz

Abstract

We describe a new algorithm for table-driven parsing with context-free grammars designed to support efficient syntactic analysis of natural language. The algorithm provides a general framework in which a variety of parser control strategies can be freely specified: bottom-up strategies, top-down strategies, and strategies that strike a balance between the two. The framework permits better sharing of parse forest substructure than other table-driven approaches, and facilitates the early termination of semantically ill-formed partial parses. The algorithm should thus find ready application to large-scale natural language processing.

Introduction

Natural language grammars are highly ambiguous. Systems for natural language understanding must therefore be carefully designed to prune out undesirable partial parses as soon as possible using semantic and pragmatic information. The need to rapidly prune a large number of unpromising parses yields two desiderata for parser design:

1. *Flexible control:* It should be possible to tailor the parser control strategy to whatever mixture of bottom-up and top-down processing facilitates both syntactic analysis and semantic interpretation as efficiently as possible.
2. *Efficiency:* Even in the best of circumstances, a parser has to consider large numbers of partial parses, so it must be able to produce these parses efficiently.

The general framework of chart parsing (Kaplan 1973) allows users to freely specify a range of parser control strategies, but inefficiently encodes the dynamic state of the parser as a large set of *edges* or *items* in a data structure called a *chart* (Kay 1986; Aho & Ullman 1977). Constructing and maintaining these edges requires a high constant-factor overhead at each stage of the parse. Table-driven algorithms such as generalized LR (GLR) parsing (Tomita 1986) or the method of Schabes (1991) aim for better efficiency by performing much of this computation off line, when the parse table is constructed. These algorithms have proven more efficient than chart parsers in practical applications, even though their time complexity may be worse in some cases.[1] Such algorithms, however, are inflexible: their control strategies are fixed and cannot be altered in any way by the user.

In this paper, we present a new framework for table-driven parsing called *L* parsing* that attempts to get the best of both worlds. Our algorithm extends the algorithm for GLR parsing. The current implementation handles arbitrary non-cyclic context-free grammars without ϵ-transitions. Like a chart parser, an L* parser allows a variety of parser control strategies to be freely specified. Like a GLR parser, an L* parser enjoys the low overhead of table-driven approaches. An L* parser also creates a concise *shared parse forest* representation of all possible parses.

We are not the first to propose a framework for freely specifying control strategies in table-driven parsing. In particular, Lang describes such a framework in Lang (1974), which is further generalized by Leermakers (1991). Our algorithm, however, makes two novel contributions that significantly increase the utility of this kind of approach in practical parsing:

1. *Improved sharing.* When Lang's algorithm is applied to parsers employing sophisticated drivers, it is often impossible to share identical subtrees because of differences in context analysis (Billot & Lang 1989; Nederhof 1993). The L* algorithm overcomes this difficulty in an efficient manner.
2. *Termination of unpromising partial parses.* An L* parser is intended to comprise only one component of a larger natural language understanding system. To maximize efficiency, it is crucial that immediately a partial parse is rejected by the rest of the system, the parser stop all work on it. As we explain below, it can be non-trivial to determine with certainty that a particular partial parse should be rejected. L* parsing provides a general mechanism for terminating partial parses in appropriate circumstances under the direction of an external oracle.

The remainder of this paper is structured as follows. We begin with a brief review of GLR parsing. We then introduce our algorithm by means of an example, and discuss sharing and termination of unpromising partial parses. Next, we describe the implementation and sketch the application of a novel parser control strategy to L* parsing with feature grammars. We conclude with a brief discussion of empirical results and plans for future work.

[1] If the length of the right-hand side of the longest rule is p, then a GLR parser has time complexity $O(n^{p+1})$ (Nederhof 1993).

GLR Parsing

GLR parsing is an extension of LR parsing that can cope with arbitrary context-free grammars.[2] Like an LR parser, a GLR parser is a shift-reduce parser, and is controlled by a deterministic finite-state driver, encoded as a *parse table*. The parse table indexes parsing actions by state and the next k input symbols. In contrast to an LR parser, however, entries in the parse table may contain multiple parsing actions. Grammatical ambiguity gives rise to table entries of this kind.

A GLR parser employs two main data structures: a *graph-structured stack* and a *packed shared parse forest*. To avoid confusion, we use the term *vertex* when referring to an element of the graph-structured stack, and the term *node* to refer to an element of the parse forest. The graph-structured stack is an extension of a basic LR stack that allows a GLR parser to deal with nondeterminism in a parse. When there are two or more possible parses, the stack splits, with a branch for each alternative. The packed shared parse forest compactly represents all of the parses of an input sentence.

A GLR parser works by repeatedly executing *reduce* and *shift* actions until an *accept* or *error* action is reached. If more than one action is specified for a given input word, any reduce actions are executed before shifts. For a more detailed description of GLR parsing, the reader is referred to Tomita & Ng (1991).

L∗ Parsing

To implement our table-driven parser with flexible control, we alter a GLR parser to allow it to *eagerly* reduce a grammar rule before its entire right-hand side (RHS) has been seen. The missing elements of the RHS are then *combined* into the parse as they arrive. We call the resulting system an L∗ parser.

Eager reduction introduces an element of top-down processing into the parser, because parent nodes can be proposed before all of their children have been recognized in the input. Exactly where eager reduction occurs can be varied, and will accordingly determine the degree of top-down processing that is performed. In the extreme case, one might eagerly reduce before seeing any RHS elements of a rule, basing the decision to reduce only on the current input word. This is equivalent to top-down parsing with a one-symbol lookahead. The opposite extreme is to never eagerly reduce, which yields a standard, bottom-up GLR parser.

Parser control strategies are specified as an input to the parse table builder. For a given grammar, a different choice of control strategy will yield a different parse table. Miller (1994) describes an algorithm for parse table construction that allows the user to specify a range of control strategies and produces parse tables for each.

Our present concern, however, is to describe the parsing algorithm, so we assume that a suitable parse table has already been constructed. The input to the algorithm is a parse table and a string of terminal symbols. The output of the algorithm is a parse forest in which common subtrees are shared and locally ambiguous structures are packed. For reasons of space, this paper does not further discuss the issue of local ambiguity packing. However, sharing of common subtrees is discussed in detail below.

The idea of the algorithm is as follows. Parsing proceeds just as for GLR parsing, except in the case of *eager reductions*—reductions performed before all input symbols corresponding to the RHS of a grammar rule have been seen. An eager reduction by a grammar rule creates an "incomplete" forest node headed by the left-hand side (LHS) of the rule, in which some or all of the children at the right-hand edge of the parse are missing. These children are *combined* into the parse one by one as they are derived from the input. After all missing children have been recognized, a second reduction called a *completing reduction* marks the forest node as complete.

An Example

To illustrate the novel aspects of the algorithm, we work through the problem of parsing a simple sentence using the toy grammar in figure 1. This grammar is annotated with control information, similar to the announce points of Abney & Johnson (unpublished manuscript): the arrow ↑ in rule 4 indicates that the parser should eagerly reduce by rule 4 upon parsing a V, even though no following NP has yet been seen in the input. Because there are no other annotations, all other rules are to be parsed bottom-up just like GLR parsing.

S → NP VP	(1)
NP → Det N	(2)
NP → N	(3)
VP → V↑ NP	(4)

Figure 1: A very small grammar

Consider the sentence "Mary ate the apple." A thumbnail sketch of the parser's actions while parsing this sentence is as follows. First it will parse "Mary" as an NP and push it onto the stack as per ordinary GLR parsing. Next, it will parse "ate" as a V and push it on the stack. The parser will then eagerly reduce by rule 4, and push the resulting incomplete VP onto the stack. This immediately triggers a further eager reduction by rule 1, creating an incomplete S. This S is incomplete even though forest nodes for each of its immediate children exist, because its child VP node is missing a child NP. The system next parses "the apple" as an NP and combines it into the incomplete VP. A completing reduction by rule 4 then marks the VP as complete. Finally, a completing reduction by rule 2 marks the S as complete.

In the remainder of this section, we trace this example in greater detail. At each step of the trace, we show

- *The graph-structured stack.* Vertices that represent stack tops are drawn as circles, with scheduled parse actions placed to the right. All other stack vertices are drawn as squares. Vertices drawn with dashed lines were created by eager reduction. Each vertex has an associated state

[2] We assume a basic familiarity with LR parsing.

(an integer) and parse forest node. Vertices have their state number inside them, and their associated parse forest node above them. The stack grows from left to right.

- *The parse forest.* Nodes in the parse forest are labelled with a grammar symbol. Dashed lines joining forest nodes represent links created by eager rather than normal reduction. A dotted line indicates the forward edge of an incomplete derivation created by eager reduction. As parsing proceeds, combine actions incrementally extend the derivation. When all RHS elements are in place, a completing reduction marks the forest node as complete, depicted by changing the dashed lines to solid ones.
- *The current word.* The word the parser is currently processing is shown in a box at the right hand edge of the diagram.

ST	ACTION							
	Det	N	V	$	*EAG*	S	VP	NP
0	s2	s1				g3		g4
1			r3	r3				
2		s11						
3				acc				
4			s5				g6	
5	e4-1, s8	e4-1, s7						g9, c4
6				r1	e1-2			
7			r3	r3				
8		s10						
9				r4	e4-2			
10			r2	r2				
11			r2	r2				

Figure 2: Parse table for grammar in figure 1

Figure 2 shows a parsing table for the grammar and parser control strategy depicted in figure 1, above. The table is indexed by state number *st* and grammar symbol *X* augmented with the special symbol *EAG*. (The function of this symbol is explained later.) An entry ACTION[*st*, *X*] is a set of parse actions. There are six kinds of action: *shift*, *reduce*, *eager-reduce*, *goto*, *combine*, and *accept*. A shift action is written "s *n*", where *n* is the state to go to after the shift. A reduce action "r *n*" specifies a reduction by the *n*-th grammar rule. An eager-reduce action "e *n-k*" specifies an eager reduction by the *n*-th grammar rule, using only the first *k* symbols of the rule's RHS. A goto action is written as "g *n*," where *n* specifies with the next state to go to. A combine action "c *n*" means the symbol *X* should be combined into the forest node (or nodes) previously created by an eager reduction by grammar rule *n*. *X* becomes the new rightmost child of this node. An accept action is written as "acc." If an ACTION entry is blank, it indicates a parse error, except in the case of the *EAG* symbol, where a blank means there are no further actions to process in this state.

We now return to the problem of parsing the sentence "Mary ate the apple." Initially the parse forest is empty, and the stack contains only a single vertex with state 0. This vertex is a stack top, so it is drawn with a circle.

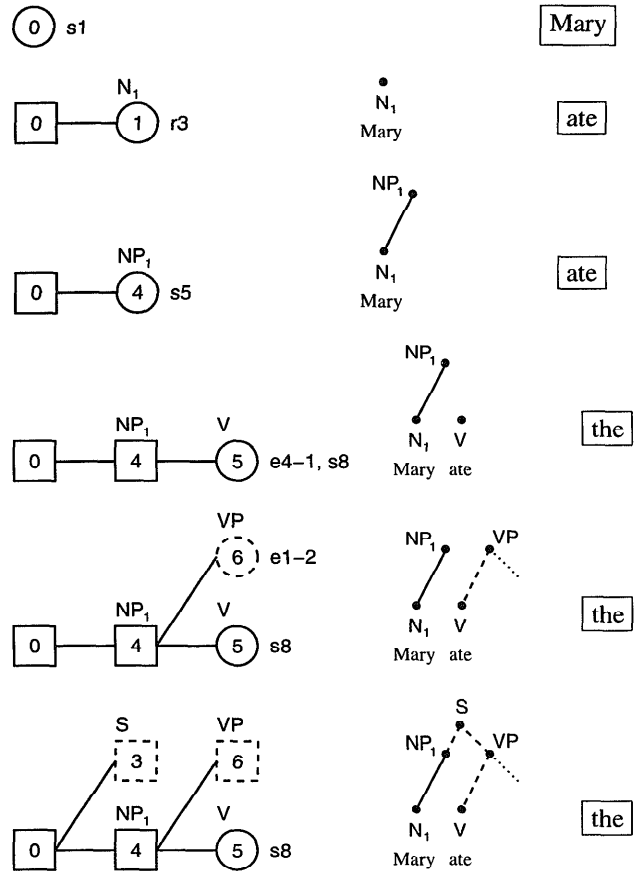

Figure 3: Trace of the eager parser

The first word to parse is "Mary." The lexical category of "Mary" is N, so the first set of parse actions to perform is ACTION[0, N] = {s1} (top diagram of figure 3).

The parser therefore performs a shift action that creates a new stack vertex with state 1 and forest node N_1. The newly created vertex becomes the stack top, so it is drawn as a circle; the original circle is redrawn as a square (second diagram of figure 3).

The next word is "ate," whose lexical category is V. ACTION[1, V] = {r3}, so the parser carries out a normal, non-eager reduction by rule 3. The vertex labeled N_1 corresponds to the RHS of rule 3 and is accordingly popped off the stack. A new vertex with state 2 is then pushed on, corresponding to the LHS of rule 3. The state of this new stack vertex is specified by ACTION[0, NP] = {g4}, because 0 is the state of the vertex at the top of the stack after N_1 was popped off. A new forest node NP_1 is also created for the new vertex, with a single child, the node N_1 (third diagram of figure 3). At this new vertex, ACTION[4, V] = {s5}, so the parser shifts the verb "ate" onto the stack, creating a new stack vertex with state 5.

The following word is "the," with lexical category Det.

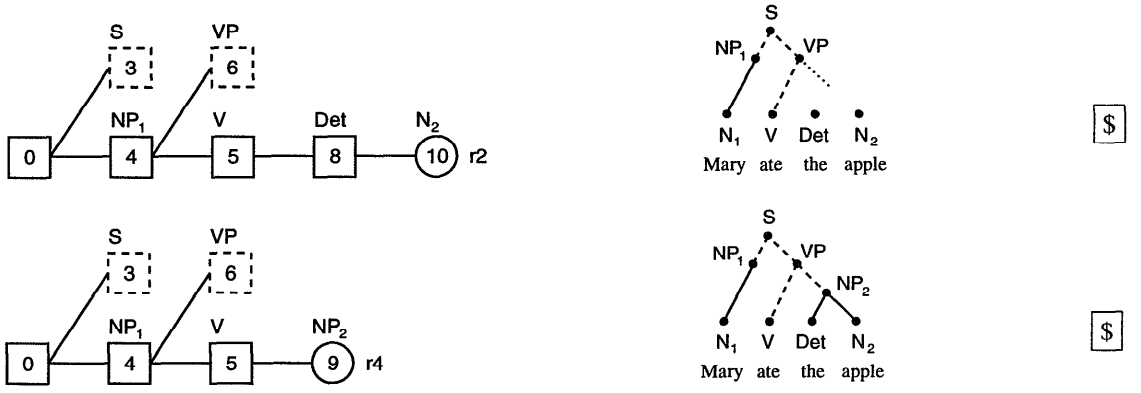

Figure 4: Trace of the eager parser (ctd)

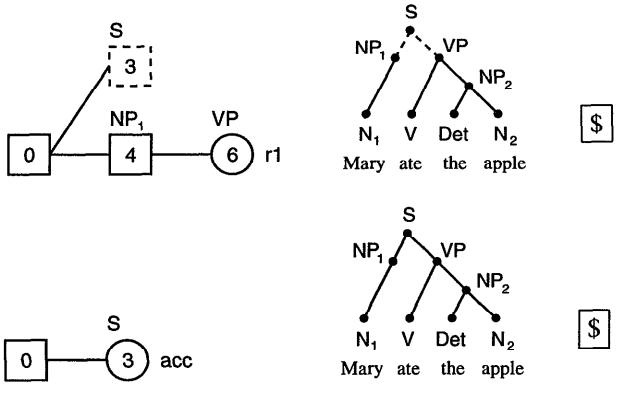

Figure 5: Trace of the eager parser (ctd)

ACTION[5, Det] = {e4-1, s8} (fourth diagram of figure 3). Reductions are processed before shifts as per GLR parsing, so the parser next eagerly reduces by rule 4, creating a new forest node VP and new stack vertex whose state is determined using the parse table entry ACTION[4, VP] = {g6}. We draw the parse tree for VP using dashed and dotted lines to indicate that the parse of the VP is not yet complete (fifth diagram of figure 3).

Having just performed an eager reduction, the system must now perform any further reductions triggered by the newly created nonterminal. We call these reductions *cascaded* reductions. A question immediately arises: which reductions should be carried out? A GLR(k) parser would use lookahead to select only those reduce actions consistent with the k input symbols that immediately follow the substring covered by the reduction. Unfortunately, these lookahead symbols are not available at the time that an eager reduction is performed, as the parse associated with the eagerly reduced rule is not yet complete. The parser therefore simply carries out all cascaded reductions that could conceivably be appropriate.

To implement this idea, we introduce a dummy symbol *EAG*. ACTION[*st*, *EAG*] contains the set of cascaded reductions to perform at state *st* after performing an eager reduction. In our example, ACTION[6, *EAG*] = {e1-2}. This reduction is eager, even though it involves the complete RHS of rule 1, because there is as yet no complete parse corresponding to the last symbol of its RHS (a VP). In fact, for this reason, all entries in the *EAG* column of the parse table are eager reductions.

The parser next carries out the eager reduction e1-2 and creates a new forest node S from NP_1 and VP, and a new stack vertex with state determined from ACTION[0, S] = {g3}. The entry ACTION[3, *EAG*] is blank, so there are no further actions to be done at this vertex (bottom diagram of figure 3).

No unprocessed reductions remain for the current word "the", so the parser performs the shift action s8, shifting "the" onto the stack. Next, the word "apple" is shifted onto the stack, making the terminator $ the current word (top diagram of figure 4). The parser then reduces Det N_2 at the top of the stack to NP_2 by rule 2. ACTION[5, NP] = {g9, c4}, so the resulting vertex has state 9. The combine action c4 is also carried out, installing NP_2 as the rightmost child of the VP (bottom diagram of figure 4).

Special data structures called *combine pointers* are maintained at stack tops to locate eagerly created derivations to combine into. A combine pointer consists of a partially completed derivation and a number representing the rule that created it by eager reduction. In general, a combine action c n at a stack top extends the derivation of each combine pointer at that stack top whose rule is n.

ACTION[9, $] = {r4} covers the same ground as the eager reduction by rule 4 carried out previously. It is therefore a "completing reduction," whose job is to indicate that the earlier eager reduction is indeed correct. Accordingly, the parser pops the RHS elements V and NP_2 off the stack, establishing VP as a new stack top, and marks its forest node as complete. This last we depict by changing the dashed lines to solid ones (top diagram of figure 5). Unlike other reductions, no new parse structure is created: all relevant structure was constructed earlier by eager reduce and combine actions. Having completed the VP node, the parser

next performs a second completing reduction, this time by rule 1, which indicates that the S is also complete. A single stack top remains, with state 3 (bottom diagram of figure 5). As ACTION[3, $] = {acc}, the parse succeeds. S is the root of the resulting parse tree.

Improving sharing in the parse forest

When a sophisticated parser control strategy is employed, the parse forests produced by existing table-driven methods such as GLR parsing and Lang's parser are not as compact as they should be. For example, Billot and Lang found that for $k > 0$, Lang's parser with an LR(k) driver often produces worse sharing than an LR(0) driver (Billot & Lang 1989). Indeed, for $k > 1$, the consequent loss of efficiency is usually so great that the overall performance of the parser degrades, even though it has a larger deterministic domain.

This problem arises because states in the parse table are expected to do double duty: states encode contextual distinctions that determine which parser actions are appropriate, while equality of states is used to decide whether or not to share subtrees. More sophisticated parsing strategies encode finer grained contextual distinctions and consequently use a larger number of states. As a result, opportunities to share substructure are lost.

We address this problem by identifying equivalence classes of states in the parse table. Two states belong to a given equivalence class if they are differentiated only by contextual distinctions that have no bearing on which parses are licensed by the left context.[3] For example, states of an LR(1) parser that are distinguished only by the lookahead of their associated items belong in the same equivalence class, because they have the same left context.

Distinct stack tops whose states are in the same equivalence class are merged into a single vertex containing a *set* of states: the union of the states of the original stack tops. The set of parse actions to perform at this new stack top is the union of the sets of parse actions specified by these states. Consequently, each reduction is carried out only once instead of several times, so the resulting parse structures are shared, improving both space and time efficiency. In this way, we can obtain the benefits of a sophisticated parser control strategy without sacrificing sharing in the parse forest.

The need to improve sharing in the parse forest is particularly acute for an L* parser, because the possibility of eager reduction introduces new contextual distinctions. It is frequently the case that one state recommends eagerly reducing by some rule, while another recommends computing the rest of the RHS of the same rule before reducing. If two such states were permitted to occupy different stack tops simultaneously, the parser would redundantly parse the subtree covered by this rule twice: once using eager reduction, and a second time using non-eager reduction. To avoid this redundant computation, such pairs of states are placed in the same equivalence class when the parse table is built. As a result, at run time the stack tops for these states are merged, causing the parser to carry out only the eager version of the parse.

[3]To be precise, two states are equivalent if their item sets contain the same dotted rules.

Terminating unpromising partial parses

Whenever a reduction or a combine action is performed, the L* parser accepts input from an oracle that signals whether or not to continue with those parses that include the newly created structure. This oracle represents a channel of communication with a larger natural language understanding system by which unpromising partial parses can be pruned. Reductions and combines can trigger rejection on the basis of mismatched grammatical features or selectional restrictions, or on the basis of pragmatic information.

In the case of non-eager reductions, the parser's response to the oracle is straightforward: if the oracle rejects a parse, the new stack top created by the offending reduction is deleted, thereby halting all further processing at that vertex.

Eager reductions and combines, however, present two special difficulties. First, it is insufficient to delete the vertex created by an eager reduction: the stack top at which associated combine actions will be performed must also be pruned. Special data structures called *kill pointers* are maintained in the stack for this purpose.

Second, a stack top at which combine actions occur cannot simply be discarded out of hand, because the forest node at this stack top may be shared by other partial derivations that have not themselves been rejected. In general, a stack top can only be rejected once every partial derivation that can conceivably use its forest node has itself been rejected. It turns out to be easy to compute the number of such partial derivations (Miller 1994); each stack top contains a counter that is initialized to this number. Every time a derivation is rejected by the oracle, the parser follows kill pointers to locate stack tops at which associated combine actions may occur, and decrements their counters. If a counter reaches zero, the associated stack top is discarded.

Implementation and experiments

The algorithm described above has been implemented in Common LISP and tested on a number of grammars, ranging from grammars specifically designed to exercise all features of L* parsing, to a grammar for a substantial subset of English. The implementation can provide a detailed trace of the actions followed during a parse: indeed, the system automatically generated the stack diagrams presented in the example above. A formal specification of the L* algorithm with and without local ambiguity packing can be found in Jones & Miller (1993).

More recently, we have extended the parser to parse using feature grammars. The feature grammar implementation has formed the basis for some initial experiments regarding the efficacy of mixed-mode parsing strategies. We have developed a general method for parse table construction that accommodates the following strategy for eager reduction: eagerly reduce if by so doing a variable binding will be established or tested. The idea is to eagerly reduce if the

reduction triggers a unification that could fail and thereby prune the parse.

Preliminary results indicate a slight degradation in average performance on grammatical sentences, but a dramatic increase in the speed with which ungrammatical sentences are rejected, if their ungrammaticality is caused by feature mismatches such as person or number disagreements. For example, using a 327-rule feature grammar, the eager reduction strategy generates 18 parse forest nodes before rejecting the following ungrammatical sentence:

(*) Jim are the best man

In contrast, ordinary GLR parsing generates 34 nodes before rejection.

Conclusions and Future Work

The L* parsing algorithm combines the low overhead of table-driven approaches with the ability to flexibly tailor the parser's control strategy to particular grammars and applications. The algorithm is strictly more general than the GLR parsing algorithm: it reduces to GLR parsing if used with a parse table that contains no eager reduce actions.

While we believe the L* parsing algorithm to be interesting in its own right, we hope that it will prove to be particularly useful and efficient for practical, large-scale natural language understanding. We hypothesize that the algorithm will achieve unusually high efficiency for three reasons. First, the algorithm is table-driven, so it automatically achieves efficiency gains over "uncompiled" approaches such as chart parsing. Second, the algorithm achieves better sharing in the parse forest than existing table-driven approaches.

Third, the L* algorithm is specifically designed to handle mixed-mode parsing strategies that may yield efficiency gains over straight bottom-up or top-down parsing. The algorithm provides a general framework in which a variety of parser control strategies can be free specified. However, unlike other general frameworks for table-driven parsing that allow a mix of bottom-up and top-down processing, L* parsing also provides a well-defined algorithm for terminating unpromising partial parses on the basis of external evidence.

Such an algorithm is essential to the implementation of efficient mixed-mode parsing strategies that balance the interacting requirements of syntactic analysis and semantic interpretation in a larger natural language understanding system. As an example of such a strategy, in Jones & Miller (1992) we advocate eagerly reducing whenever the reduction is likely to generate semantic preferences that provide evidence against unpromising partial parses. Unpromising partial parses can then be pruned, increasing overall efficiency of the natural language processing system. Our experiments with feature grammars constitute a first tentative step towards validating this kind of parsing strategy: it is reasonable to suppose that selectional restrictions can be used to quickly rule out many semantically ill-formed partial parses in much the same way that grammatical features can serve to efficiently rule out ungrammatical sentences.

The next phase of our research will employ the L* framework to further explore and validate a variety of mixed-mode parsing strategies, including the ones sketched above and a number of other strategies proposed in the literature. For example, Steel and De Roeck claim that certain phenomena such as traces are most efficiently analyzed top down, while others such as coordinate conjunctions should be parsed bottom up (Steel & De Roeck 1987). The L* algorithm provides a convenient uniform framework for comparing the relative efficiency of different approaches. We also plan to change the search within the parser from a breadth-first to a best-first mechanism, and extend the oracle to allow graded judgements that can inform this search.

References

Abney, S. P., and Johnson, M. Memory requirements and local ambiguity of parsing strategies. Unpublished manuscript.

Aho, A. V., and Ullman, J. D. 1977. *Principles of Compiler Design*. Addison-Wesley.

Billot, S., and Lang, B. 1989. The structure of shared forests in ambiguous parsing. In *Proc. 27th Annual Meeting of the ACL*, 143–151. Vancouver, British Columbia: ACL.

Jones, E. K., and Miller, L. M. 1992. Eager GLR parsing. In *First Australian Workshop on Natural Language Processing and Information Retrieval*.

Jones, E. K., and Miller, L. M. 1993. The L* parsing algorithm. Technical Report CS-TR-93/9, Victoria University of Wellington.

Kaplan, R. M. 1973. A general syntactic processor. In Rustin, R., ed., *Natural Language Processing*. Algorithmic Press. 193–241.

Kay, M. 1986. Algorithm schemata and data structures in syntactic processing. In Grosz, B. J.; Jones, K. S.; and Webber, B. L., eds., *Readings in Natural Language Processing*. Morgan Kaufmann. 35–70.

Lang, B. 1974. Deterministic techniques for efficient non-deterministic parsers. In Loeck, J., ed., *Proc. 2nd Colloquium on Automata, Languages and Programming*, volume 14 of Lecture Notes in Computer Science, 255–269. Saarbrücken: Springer-Verlag.

Leermakers, R. 1989. How to cover a grammar. In *Proc. 27th Annual Meeting of the ACL*, 135–142. Vancouver, British Columbia: ACL.

Miller, L. M. 1994. Flexible table-driven parsing for natural language understanding. Master's thesis, Victoria University of Wellington. Forthcoming.

Nederhof, M.-J. 1993. Generalised left-corner parsing. In *Proc. 6th Conference of the European Chapter of the ACL*, 305–314. Utrecht, The Netherlands: ACL.

Schabes, Y. 1991. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *Proc. 29th Annual Meeting of the ACL*, 106–113. Berkeley, California: ACL.

Steel, S., and De Roeck, A. N. 1987. Bidirectional chart parsing. In Mellish, C. S., and Hallam, J., eds., *Advances in Artificial Intelligence (Proc. AISB-87)*, 223–235. J. Wiley and Sons.

Tomita, M., and Ng, S. 1991. The generalized LR parsing algorithm. In Tomita, M., ed., *Generalized LR Parsing*. Kluwer Academic Publishers. 1–16.

Tomita, M. 1986. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers.