VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

The L∗ Parsing Algorithm

Eric K. Jones and Linton M. Miller

Technical Report CS-TR-93/9
December 1993

Abstract

This report describes a new algorithm for table-driven parsing with arbitrary context-free grammars. The algorithm provides a 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 algorithm thus combines the flexibility of chart parsing with the low overhead of table-driven approaches, and should find ready application to natural language processing.

Publishing Information

A shorter version of this report has been accepted for the *17th Annual Computer Science Conference* to be held in Christchurch, New Zealand, January 1994.

Author Information

Eric Jones is on the academic staff of the Department of Computer Science at Victoria University of Wellington and a member of the department's artificial intelligence group.

Linton Miller is a Masters student with the Department of Computer Science at Victoria University of Wellington.

1 Introduction

Parsers for general context-free grammars have found wide application in natural language processing, because grammars for natural languages are close to context-free [4]. To be useful in natural language processing, however, a parsing algorithm must cope well with grammatical ambiguity, as natural language grammars are highly ambiguous. The number of valid syntactic parses of a natural language sentence can be larger than exponential in the length of the sentence [9], and sentences of average length occurring in newspaper stories often admit hundreds of parses [5]. Most of the time, however, only one of these many parses is consistent with context and the meanings of the words in the sentence. Systems for natural language understanding must therefore be carefully designed to prune out undesirable partial parses as soon as possible using semantic information.

The need to rapidly prune a large number of unpromising parses yields two desiderata for parser design:

1. *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. Table-driven algorithms such as generalised LR (GLR) parsing [14] or the method of Schabes [11] are significantly more efficient in practical applications than other approaches such as chart parsing, even though they may have the same worst-case complexity.

2. *Flexible control:* It should be possible to tailor the parser control strategy to whatever mixture of bottom-up and top-down processing allows the parser to facilitate both syntactic analysis and semantic interpretation as efficiently as possible. Pure bottom-up parsers are unnecessarily inefficient because they often fail to allow semantic information to be brought to bear as early as practicable in processing to prune unpromising partial parses [6]. Pure top-down parsers are likewise inefficient, because they explore syntactic hypotheses that are entirely unsupported by the input. Moreover, certain natural language phenomena such as traces are most efficiently analysed top down, while others such as coordinate conjunctions are best parsed bottom up [13].

   Ideally, a parser should be designed to balance top-down and bottom-up processing so as to minimise the total effort of the natural language processing system as a whole. The general framework of chart parsing [7] allows users to freely specify a range of parser control strategies and is consequently often the method of choice in real-world applications.

Existing parsers exhibit a tradeoff between efficiency and flexibility of control. Chart parsers encode their dynamic state as a large set of "edges" [8] or "items" [2] in a data structure called a *chart*. Maintaining these edges requires a high constant-factor overhead at each stage of the parse.

By contrast, table-driven approaches such as GLR parsing perform much of this computation off line, when the parse table is constructed. Indeed, a GLR parser can be viewed as a "compiled" version of a left-corner chart parser that constructs fewer edges (except in pathological cases). It follows that a GLR parser is potentially more efficient than even the most carefully designed left-corner chart parser. A GLR parser also incorporates a *top-down filter*: that is, it never carries out parsing actions that are inconsistent with the left context. Nevertheless, the control strategy of a GLR parser is rigidly specified and cannot be tailored to fit the properties of particular grammars or systems for natural language processing.

In this report, we present a table-driven algorithm for syntactic analysis that allows a user to specify parsing strategies that combine bottom-up and top-down processing. Our algorithm modifies and extends the algorithm for GLR parsing. Like chart parsing, our algorithm constitutes a general framework in which a variety of parser control strategies can be freely specified. Like GLR parsing, the algorithm automatically incorporates a top-down filter and enjoys the low overhead of table-driven approaches. Finally, the algorithm develops a space-efficient "shared packed forest" representation of all possible parses.

The remainder of this report is structured as follows. Section 2 briefly reviews GLR parsing. Section 3 describes our algorithm by means of an example. Section 4 concludes with a brief discussion of the advantages and limitations of our approach.

2 GLR Parsing

GLR parsing is an extension of LR parsing that can cope with arbitrary context-free grammars.[1] Like an LR parser, a GLR parser is a *shift-reduce* parser: elements of the right-hand side (RHS) of a grammar rule are shifted one by one on to a stack as they are recognized in the input. Complete RHSs are then reduced, or replaced by their corresponding left-hand sides (LHSs), whenever specified by the parse table.

A GLR parser 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. An identical driver is used to control LR parsers. In contrast to an LR parser, however, parse table entries 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 *shared packed 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 shared packed forest compactly represents all the possible parses of an input sentence. The forest is constructed such that all common subtrees are *shared*, and represented only once in the forest. Subtrees that represent different parses of the same input substring are *packed* into a common root node that has multiple lists of children, one for each parse. This operation is called *local ambiguity packing.*

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.

Whenever two or more reductions are specified that cover the same input substring and that result in the same state, local ambiguity packing is appropriate. In this case, each reduction corresponds to an alternative parse of the substring headed by the same nonterminal.

For a more detailed description of GLR parsing, the reader is referred to [15].

3 Table-Driven 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 RHS has been seen. The missing elements of the RHS are then "combined" into the parse as they arrive. We call the resulting system a 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 can imagine eagerly reducing 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. In [10], we describe an algorithm

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

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 is a collection of parse trees in which common subtrees are shared. 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" parse node headed by the 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 parse node as complete.

3.1 An Example

To illustrate 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: the arrow $\uparrow$ 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.

$$
\begin{array}{llr}
\mathrm{S} & \rightarrow \mathrm{NP\ VP} & (1) \\
\mathrm{NP} & \rightarrow \mathrm{det\ N} & (2) \\
\mathrm{NP} & \rightarrow \mathrm{N} & (3) \\
\mathrm{VP} & \rightarrow \mathrm{V}\!\uparrow \mathrm{NP} & (4)
\end{array}
$$

Figure 1: A very small grammar

Consider the sentence "John saw the man." A thumbnail sketch of the parser's actions while parsing this sentence is as follows. First it will parse "John" as a NP and push it onto the stack as per ordinary GLR parsing. Next, it will parse "saw" 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 parse nodes for each of its immediate children exist, because its child VP node is missing a child NP. The system next parses "the man" 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. All other stack vertices are drawn as squares. Vertices drawn with dashed lines were created by eager reduction. Each vertex has an associated state (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, which possibly has a subscript to distinguish different nodes labelled by the same 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. Incomplete derivations are created

STATE	ACTION							
	Det	N	V	$	*EAG*	S	NP	VP
0	s4	s3				g1	g2	
1				acc				
2			s6					g7
3			r3	r3				
4		s5						
5			r2	r2				
6	e4-1, s9	e4-1, s8					g10, c4	
7				r1	e1-2			
8			r3	r3				
9		s11						
10				r4	e4-2			
11			r2	r2				

Figure 2: Parse table for grammar in figure 1

by eager reduction. As parsing proceeds, combine actions incrementally extend them. When all RHS elements are in place, a completing reduction marks the the forest node as complete, depicted by changing the dashed lines to solid ones.

- *The current word.* The word of the sentence the parser is currently working with is shown in a box at the right hand edge of the diagram.
- *Next parse actions.* These are entries from the parse table, and are placed to the right of the stack tops.

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 parse node 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 "John saw the man." 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. The first word to parse is "John." The lexical category of "John" is N, so the first set of parse actions to perform is ACTION[0, N] = {`s3`} (top diagram of figure 3).

We therefore perform a shift action that creates a new stack vertex with state 3 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 (middle diagram of figure 3).

The next word is "saw," whose lexical category is V. ACTION[3, V] = {`r3`}, so we carry 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] = {`g2`}, 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

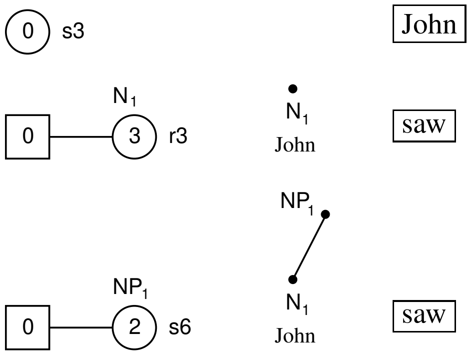

Figure 3: Trace of the eager parser

also created for the new vertex, with a single child, the node N_1 (bottom diagram of figure 3). At this new vertex, ACTION[2, V] = {`s6`}, so we shift the verb "saw" onto the stack, creating a new stack node with state 6.

The following word is "the," with lexical category Det. ACTION[6, Det] = {`e4-1, s9`} (top diagram of figure 4). Reductions are processed before shifts as per GLR parsing, so the parser next eagerly reduces by rule 4 using only the first symbol on the right-hand side of the rule, creating a new stack vertex with forest node VP and state 7. This state is determined using the parse table entry ACTION[2, VP] = {`g7`}. We draw the parse tree for VP with dashed and dotted lines to indicate that

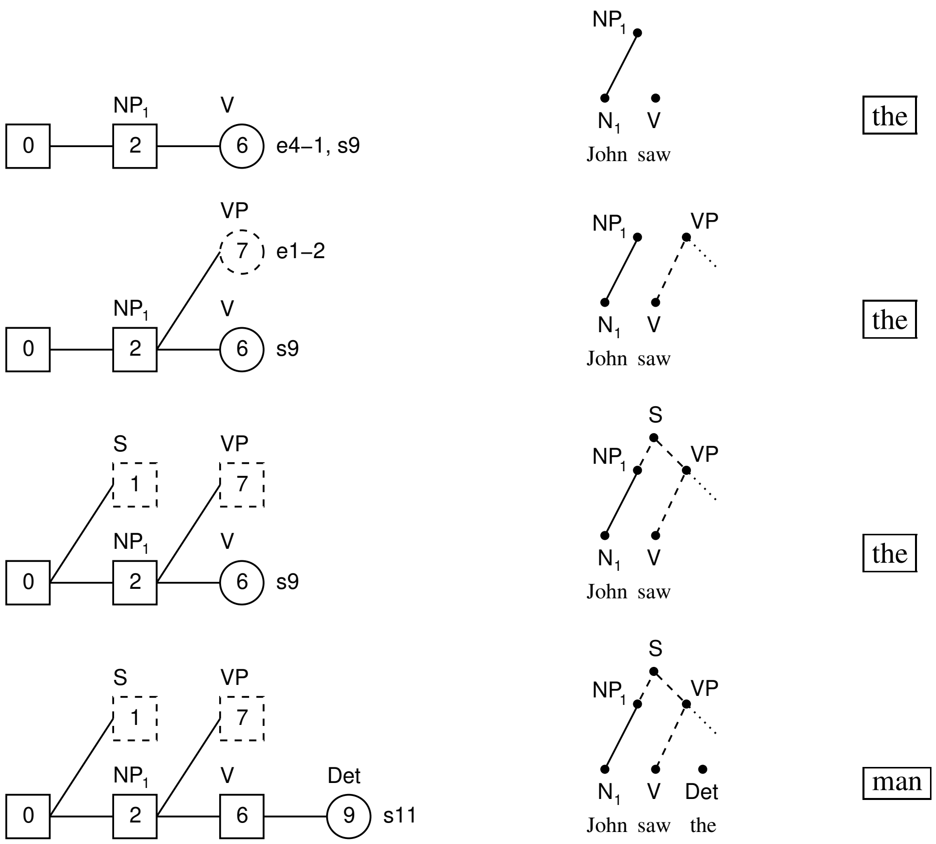

Figure 4: Trace of the eager parser (ctd)

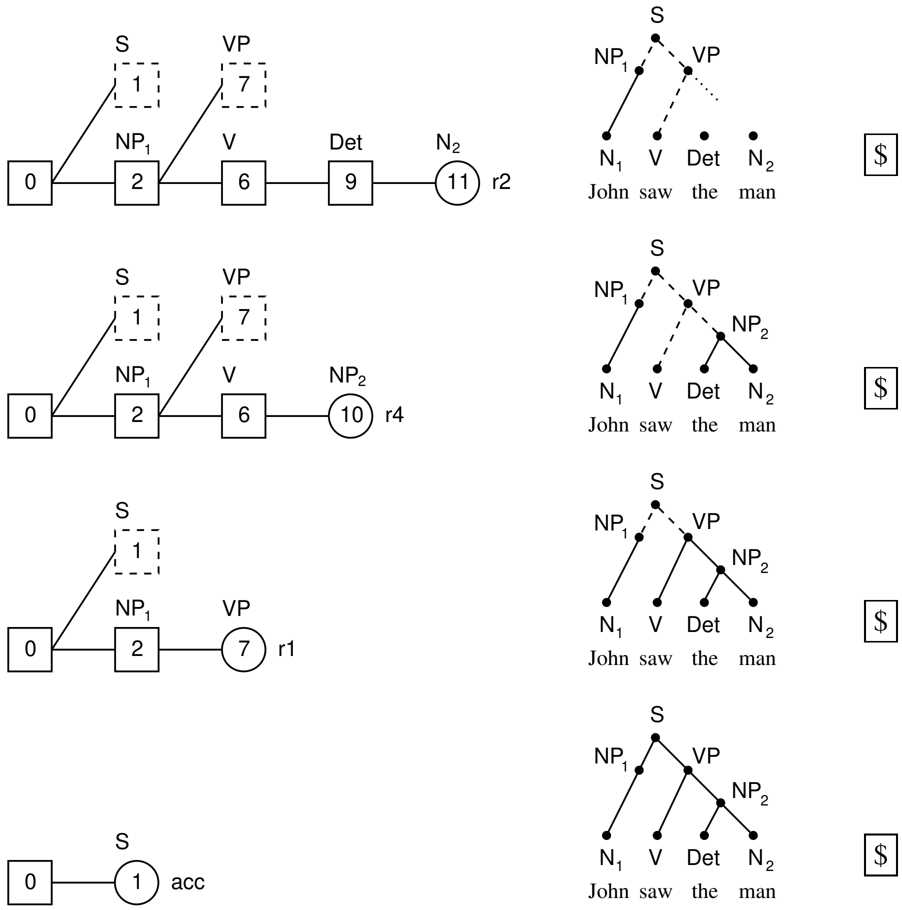

Figure 5: Trace of the eager parser (ctd)

the parse of the VP is not yet complete (second diagram of figure 4).

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[7, *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] = $\{$`g1`$\}$. The entry ACTION[1, *EAG*] is blank, so we do nothing further at this vertex (third diagram of figure 4).

No unprocessed reductions remain for the current word ("the"), so the parser finally performs the shift action `s9`, shifting "the" onto the stack. The current word is now the noun "man" (bottom diagram of figure 4). This word is shifted onto the stack, making the terminator $ the current word (top diagram of figure 5). Next, the parser reduces Det N_2 at the top of the stack to NP_2 by rule 2.

ACTION[6, NP] = $\{$`g10, c4`$\}$, so the new vertex resulting from this reduction has state 10. The combine action also means that forest node NP_2 is combined into the VP created previously during eager reduction by rule 4. The combine action installs NP_2 as the rightmost child of the VP, but leaves the stack unchanged (second diagram of figure 5).

ACTION[10, $] = $\{$`r4`$\}$ covers the same ground as the eager reduction by rule 4 carried out previously. It is therefore a "completing reduction," whose job it 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 (third 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 1 (bottom diagram of figure 5). As ACTION[1, $] = $\{$`acc`$\}$, the parse succeeds. S is the root of the resulting parse tree.

3.2 Local Ambiguity Packing

The example described above does not involve the problem of local ambiguity packing. Local ambiguity packing can reduce the amount of stack and parse forest structure created by an exponential factor and has been shown to yield substantial efficiency gains in practice [12]. Our current strategy for packing is derived from the approach followed by Tomita [14], which packs different derivations that cover the same input substring by creating a single common parent. For grammars in Chomsky Normal Form it can be shown that a GLR parser using Tomita's packing algorithm has worst case complexity $O(n^3)$ as a function of the length of the input sentence, which is comparable to the most efficient algorithms used in practice.

More generally, any context-free grammar can be easily transformed into a grammar in Chomsky Normal Form that generates the same language [1]. Billot and Lang [3] describe an optimal packing strategy for arbitrary context-free grammars similar to this transformation procedure, which could be easily incorporated into our current algorithm.

Whichever packing strategy is chosen, eager reduction makes local ambiguity packing difficult to implement. At the time that an eager reduction is performed, an accurate determination of whether or not to pack an eagerly created derivation with other derivations cannot always be made, because the parser does not yet know the entire substring that the eager reduction covers. In particular, two eager reductions to a given nonterminal and state may or may not complete at the same future input symbol. Our solution to this problem is to provisionally pack eager reductions, then unpack them later if they turn out to span different substrings.

One thing we do know for certain, however: eager reductions at a given input symbol never pack with non-eager reductions. The eager reduction must cover at least one more input symbol than the non-eager one, so the corresponding derivations must cover different substrings and should therefore not be packed together.

3.3 Implementation

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.

Detailed descriptions of the parsing algorithm are given in appendices. Appendix A describes a version of the algorithm without local ambiguity packing. Appendix B gives the complete version of the algorithm.

4 Conclusions and Future Work

Our table-driven 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 our algorithm to be interesting in its own right, we hope that it will prove to be a particularly efficient practical parsing algorithm for large-scale natural language understanding. We hypothesise that the algorithm will achieve unusually high efficiency because it is table driven but also accommodates parser control strategies that may yield efficiency gains over straight bottom-up or top-down parsing. For example, in [6] we advocate a parsing strategy that aims to minimise overall search within a natural language understander by eagerly reducing whenever the reduction is likely to generate semantic preferences that resolve local ambiguities. We are currently conducting experiments to test our hypothesis that strategies such as this can improve performance. We also plan to investigate improving performance by changing the search within the parser from a breadth-first to a best-first mechanism.

Table-driven L$*$ parsing is not without disadvantages. Only left-to-right parsing strategies are permitted, so it is not as general a framework as chart parsing. Perhaps more seriously, the algorithm is intolerant of ungrammatical input. However, this problem is common to all parsers that are driven by an LR automaton (or that maintain a top-down filter). Although there may be ways to cope with ungrammaticality while retaining a table-driven approach, "uncompiled" frameworks such as chart parsing may be more suitable. More generally, efficient parsing in the presence of ungrammatical input remains an important open problem.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, 1972.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[3] Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, Vancouver, British Columbia, 1989. ACL.

[4] Gerald Gazdar and Chris Mellish. *Natural Language Processing in Prolog*. Addison-Wesley, 1989.

[5] P. S. Jacobs, G. R. Krupka, and L. F. Rau. Lexico-semantic pattern matching as a companion to parsing in text understanding. In *Proceedings of the Speech and Natural Language Workshop*, pages 337–341, Pacific Grove, California, 1991. Morgan Kaufmann.

[6] Eric K. Jones and Linton M. Miller. Eager GLR parsing. In *First Australian Workshop on Natural Language Processing and Information Retrieval*, Melbourne, Australia, November 1992.

[7] Ronald M. Kaplan. A general syntactic processor. In Randall Rustin, editor, *Natural Language Processing*, pages 193–241. Algorithmic Press, 1973.

[8] Martin Kay. Algorithm schemata and data structures in syntactic processing. In Barbara J. Grosz, Karen S. Jones, and Bonnie L. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufmann, 1986.

[9] Hiroshi Maruyama. Structural disambiguation with constraint propagation. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, pages 31–38, Pittsburgh, Pennsylvania, 1990. ACL.

[10] Linton M. Miller. Flexible table-driven parsing for natural language understanding. Master's thesis, Victoria University of Wellington, 1994. (In preparation.)

[11] Yves Schabes. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pages 106–113, Berkeley, California, June 1991. ACL.

[12] Patrick Shann. Experiments with GLR and chart parsing. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 17–34. Kluwer Academic Publishers, 1991.

[13] Sam Steel and Anne N. De Roeck. Bidirectional chart parsing. In Christopher S. Mellish and John Hallam, editors, *Advances in Artificial Intelligence (Proceedings of AISB-87)*, pages 223–235. J. Wiley and Son, 1987.

[14] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.

[15] Masaru Tomita and See-Kiong Ng. The generalized LR parsing algorithm. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 1–16. Kluwer Academic Publishers, 1991.

Appendix A: Algorithm L$*$ Without Packing

The following is a formal specification of the L$*$ parsing algorithm without local ambiguity packing. A complete version of the algorithm incorporating local ambiguity packing is given in Appendix B.

One important aspect of the algorithm merits explicit mention, as it is not discussed in the main body of the text. The algorithm makes use of special data structures called *combine pointers* to keep track of vertices created by eager reductions and their associated derivations. Combine pointers are stored at stack vertices and are used in two ways. First, `combine` actions follow combine pointers to locate derivations to combine into. Second, completing reductions at a stack top follow combine pointers to locate vertices that correspond to completed derivations. These vertices become new stack tops at which further reductions and shifts can take place.

Input

A parse table ACTION[`state`, `symbol`] for the context-free grammar $G = \langle N, \Sigma, R, S\rangle$ and an input string $z \in \Sigma^*$. Entries in the parse table are sets of parsing actions. Each action has the form "`shift` s", "`reduce` r", "`eager-reduce` r, k", "`goto` s", "`combine` r", or "`accept`". N is a set of non-terminals, Σ is a set of terminals, R is a set of grammar rules of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup \Sigma)^*$, and S is the start symbol. The state s_0 is designated as the start state.

Output

A list of root nodes of a shared parse forest for z if $z \in L(G)$, otherwise an error indication.

Data Structures

A *vertex* in the graph-structured stack is a tuple $\langle s, n, \mathcal{C}, \mathcal{S}\rangle$, where s is a state, n is a forest node, $\mathcal{C}$ is a set of combine pointers, and $\mathcal{S}$ is the set of successor vertices in the stack. For notational convenience, the elements of a vertex v can be referenced using the functions State(v), Node(v), Combine-Ptrs(v), and Successors(v).

A *node* in the shared forest is a tuple $\langle X, d\rangle$, where $X \in N$, and d is a list of child forest nodes. The elements of a node n can be referenced using the functions Nonterm(n) and Deriv(n).

A *combine pointer* is a pair $\langle r, \mathcal{V}\rangle$, where $r \in R$ and $\mathcal{V}$ is a set of vertices. The elements of a combine pointer c can be referenced using the functions Rule(c), and Vertices(c).

A *path* is a contiguous sequence of vertices $(v_1, \ldots, v_k)$ in the stack. That is, for $i = 2, \ldots, k$, $v_i \in$ Successors(v_{i-1}).

FRONTIER stores a set of pairs $\langle v, a\rangle$, where v is a vertex and a is a parse action yet to be performed at v. The vertices within the pairs of this list form the active stack tops.

ω denotes the current input symbol.

Main Loop

· Add a terminator symbol \$ to the end of the input string z	
· $\omega \leftarrow$ The first symbol of z	
· $v_0 \leftarrow \langle s_0, \text{NIL}, \{\}, \{\}\rangle$	· Initialise the stack
· Call **Schedule**(v_0, ω)	
· Loop	· Perform reductions followed by shifts until acceptance or rejection.
· Call **Reduce**()	
· Call **Shift**()	
· If FRONTIER contains only pairs of the form $\langle v_i$, "`accept`"$\rangle$ then halt and return $\{$Node$(v_i) \mid \langle v_i$, "`accept`"$\rangle \in$ FRONTIER$\}$	
· If FRONTIER = $\{\}$ then halt and signal an error.	

Reduce()

Perform all outstanding reductions by calling the subroutine appropriate to each reduce action. Non-eager reduce actions are processed first, followed by eager reduce actions.

· While $\exists x \in$ FRONTIER of the form $\langle v, \text{“}\texttt{reduce}\ X \rightarrow \alpha\text{”}\rangle$:
  · Remove x from FRONTIER
  · $\mathcal{C} \leftarrow \{c \in \textsf{Combine-Ptrs}(v) \mid \textsf{Rule}(c) = X \rightarrow \alpha\}$
  · If $\mathcal{C} \neq \{\}$ then
    · Call **Completing-Reduce**$(v, \mathcal{C})$
  Else
    · $\mathcal{P} \leftarrow \{p \mid p$ is a path of length $|\alpha|$ starting at v in the stack$\}$
    · $\forall p \in \mathcal{P}$, call **Full-Reduce**$(p, X \rightarrow \alpha)$
· While $\exists x \in$ FRONTIER of the form $\langle v, \text{“}\texttt{eager-reduce}\ r, k\text{”}\rangle$:
  · Remove x from FRONTIER
  · $\mathcal{P} \leftarrow \{p \mid p$ is a path of length k starting at v in the stack$\}$
  · $\forall p \in \mathcal{P}$, call **Eager-Reduce**$(p, r)$

· Process all outstanding non-eager reductions at stack tops.

· Each element of $\mathcal{C}$ corresponds to a previous eager reduction that is now complete.

· Process all outstanding eager reductions at stack tops.

Completing-Reduce$(v, \mathcal{C})$

Perform a completing reduction at vertex v. As this reduction covers the same ground as a previous eager reduction, there is no new parse structure to create, and all that needs to be done is schedule any actions at the vertices associated with the result of the completing reduction.

· $\forall c \in \mathcal{C}$
  · Remove c from $\textsf{Combine-Ptrs}(v)$
  · $\forall v' \in \textsf{Vertices}(c)$
    · Call **Schedule**(v', ω)

· $\mathcal{C}$ is a set of combine pointers that point to the vertices created earlier by an eager reduction that is now being completed.

Full-Reduce$(v_1 \ldots v_k, X \rightarrow \alpha)$

Non-eagerly reduce by the rule $X \rightarrow \alpha$, creating a new forest node X whose children are the nodes of vertices $v_1 \ldots v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by this reduction.

· $n' \leftarrow \langle X, (\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1))\rangle$	· Create a new forest node.
· $\Pi \leftarrow$ A partition of $\mathsf{Successors}(v_k)$ by goto value on symbol X	· v belongs to the set $\pi_s \in \Pi$ if and only if "`goto` s" $\in$ ACTION[$\mathsf{State}(v)$,X]. See figure 1(a).
· $\forall \pi_s \in \Pi$ · $v' \leftarrow \langle s, n', \{\}, \pi_s\rangle$	· Create new vertices containing the new forest node, one for each element of Π.
· $\forall v \in \pi_s$ · $\forall a \in$ ACTION[$\mathsf{State}(v), \omega$] s.t. a = "`combine` r" · Call **Combine**(v, v', r) · Call **Schedule**(v', ω)	· Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have $\mathsf{Node}(v)$ as their rightmost element.

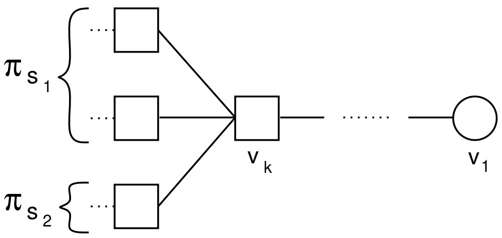

Figure 1(a): Stack before full reduction

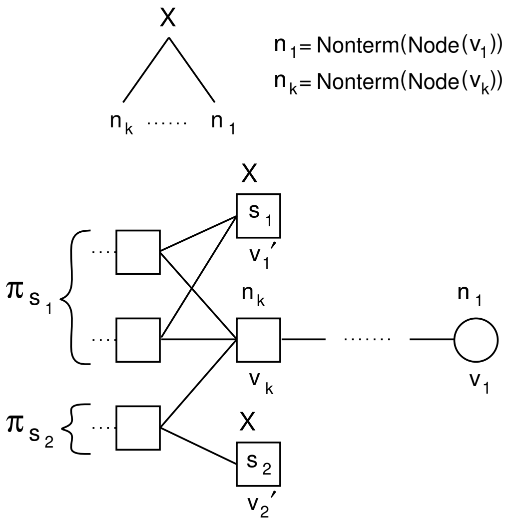

Figure 1(b): Stack and new forest node after full reduction

Eager-Reduce$(v_1 \ldots v_k, X \rightarrow \alpha)$

Eagerly reduce by the rule $X \rightarrow \alpha$, creating a new incomplete forest node X whose children are the nodes of vertices $v_1 \ldots v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by reduction.

· $n' \leftarrow \langle X, (\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1))\rangle$	· Create a new forest node.
· $c \leftarrow \langle X \rightarrow \alpha, \{\}\rangle$	· Create a new combine pointer.
· $\Pi \leftarrow$ A partition of Successors(v_k) by goto value on symbol X	· v belongs to the set $\pi_s \in \Pi$ if and only if "`goto` s" $\in$ ACTION[State(v),X]. See figure 2(a).
· $\forall \pi_s \in \Pi$	· Create new vertices containing this new node, one for each element of Π.
· $v' \leftarrow \langle s, n', \{\}, \pi_s\rangle$	
· Add v' to Vertices(c)	
· $\forall v \in \pi_s$	· Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.
· $\forall a \in$ ACTION[State(v), ω] s.t. a = "`combine` r"	
· Call **Combine**(v, v', r)	
· Call **Schedule**$(v'$, *EAG*$)$	
· Add c to Combine-Ptrs(v_1)	

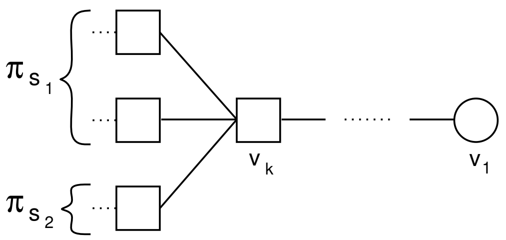

Figure 2(a): Stack before eager reduction

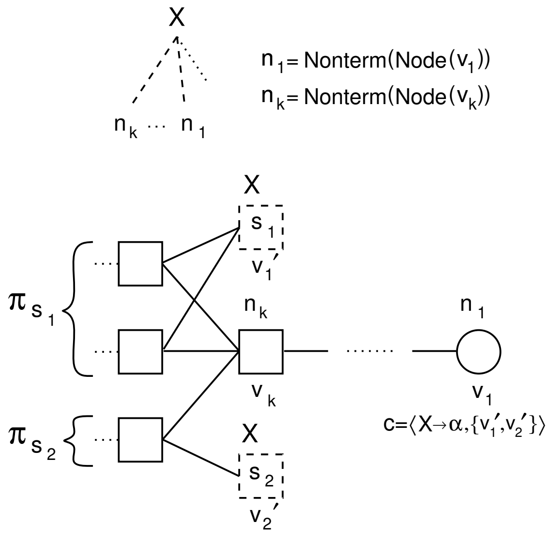

Figure 2(b): Stack and new forest node after eager reduction

Combine(v_f, v_t, r)

Combine Node(v_t) into partial derivations created by eager reduction whose rightmost element is currently Node(v_f). These derivations are identified by the combine pointers associated with v_f.

· $\forall c \in$ Combine-Ptrs(v_f) s.t. Rule$(c) = r$
  · $v \leftarrow$ An arbitrary element of Vertices(c)
  · Add Node(v_t) to the end of Deriv(Node(v))
  · Remove c from Combine-Ptrs(v_f)
  · Add c to Combine-Ptrs(v_t)

· All elements of Vertices(c) have the same forest node.
· Move the combine pointers forward to v_t so that further combines (or completing reductions) can be performed there.

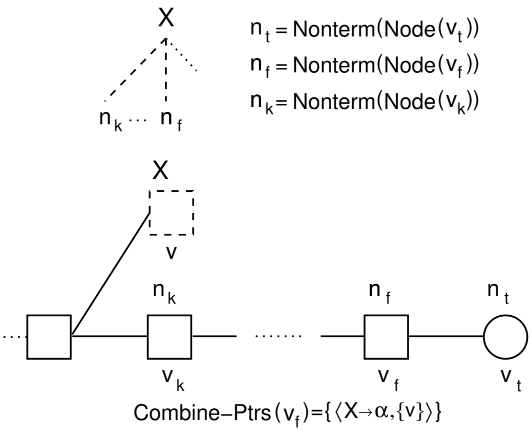

Figure 3(a): Stack and forest node before combine

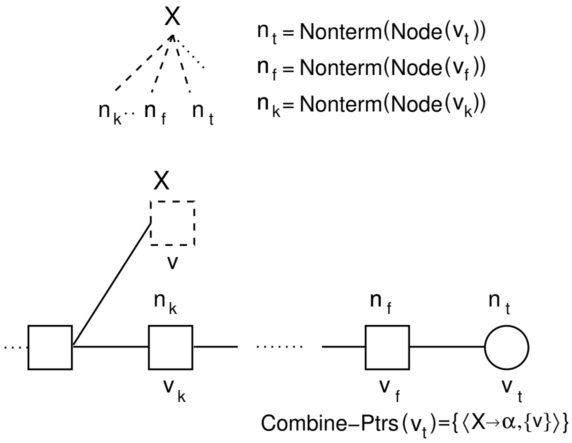

Figure 3(b): Stack and forest node after combine

Shift()

Shift the next terminal symbol onto all the stack tops and create a new node for it in the parse forest. Combine this new node into other partial derivations created previously by eager reduction, and schedule any actions triggered by the shift.

- $n \leftarrow \langle \omega, \{\} \rangle$
- $\omega \leftarrow$ The next symbol of the input string
- $\mathcal{S} \leftarrow \{\langle v, a \rangle \in$ FRONTIER $\mid a =$ "`shift` $s''\}$
- FRONTIER $\leftarrow$ FRONTIER $- \mathcal{S}$
- $\Pi \leftarrow$ partition of $\mathcal{S}$ according to goto state of the `shift` actions
- $\forall \pi_s \in \Pi$
  - $\mathcal{V} \leftarrow \{v \mid \langle v, a \rangle \in \pi_s\}$
  - $v_s \leftarrow \langle s, n, \{\}, \mathcal{V} \rangle$
  - $\forall x \in$ FRONTIER of the form $\langle v,$ "`combine` r" $\rangle$ s.t. $v \in \mathcal{V}$
    - Remove x from FRONTIER
    - Call **Combine**(v, v_s, r)
  - Call **Schedule**(v_s, ω)

- Create a new forest node for the shifted input symbol.
- $\mathcal{S}$ is the set of all shift actions to perform.
- Each $\pi_s \in \Pi$ consists of elements of the form $\langle v,$ "`shift` s" $\rangle$.
- Create a new vertex for each member of Π.
- Combine the newly created n into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.

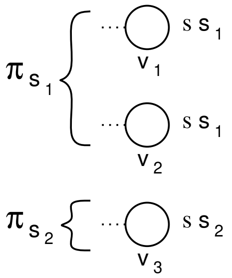

Figure 4(a): Stack tops with outstanding shift actions

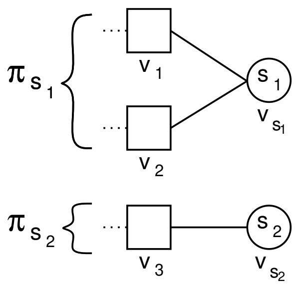

Figure 4(b): Stack after shifting

Schedule(v, X)

Add to FRONTIER all possible actions to be performed at vertex v.

- $\mathcal{A} \leftarrow \{a \in$ ACTION[State$(v), X] \mid$
  $(a =$ "`eager-reduce` r, k" $\Rightarrow$
  $\neg\exists c \in$ Combine-Ptrs(v) s.t. Rule$(c) = r)$
- $\forall a \in \mathcal{A}$, add $\langle v, a \rangle$ to FRONTIER

- Schedule all actions except eager reductions that repeat an eager reduction carried out earlier.

□

Appendix B: Algorithm L$*$

The following is a formal specification of the complete L$*$ parsing algorithm.

Input

A parse table ACTION[`state`, `symbol`] for the context-free grammar $G = \langle N, \Sigma, R, S\rangle$ and an input string $z \in \Sigma^*$. Entries in the parse table are sets of parsing actions. Each action has the form "`shift` s", "`reduce` r", "`eager-reduce` r, k", "`goto` s", "`combine` r", or "`accept`". N is a set of non-terminals, Σ is a set of terminals, R is a set of grammar rules of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup \Sigma)^*$, and S is the start symbol. The state s_0 is designated as the start state.

Output

The root node of a packed shared parse tree for z if $z \in L(G)$, otherwise an error indication.

Data Structures

A *vertex* in the graph-structured stack is a tuple $\langle s, n, l, \mathcal{C}, \mathcal{S}\rangle$, where s is a state, n is a forest node, l is the length of the longest path back to the base of the stack, $\mathcal{C}$ is a set of combine pointers, and $\mathcal{S}$ is the set of successor vertices in the stack. For notational convenience, the elements of a vertex v can be referenced using the functions State(v), Node(v), Depth(v), Combine-Ptrs(v), and Successors(v).

A *node* in the shared forest is a tuple $\langle X, \mathcal{D}\rangle$, where $X \in N$, and $\mathcal{D}$ is a set of derivations. The elements of a node n can be referenced using the functions Nonterm(n) and Derivs(n).

A *derivation* is a pair $\langle c, x\rangle$, where c is a list of child forest nodes and $x \in \{$`complete, incomplete`$\}$ indicates whether the derivation is complete or not. The elements of a derivation d can be referenced using the functions Children(d), and Status(d).

A *combine pointer* is a tuple $\langle r, d, \mathcal{V}\rangle$, where $r \in R$, d is a derivation and $\mathcal{V}$ is a set of vertices. The elements of a combine pointer c can be referenced using the functions Rule(c), Deriv(c), and Vertices(c).

A *path* is a contiguous sequence of vertices $(v_1, \ldots, v_k)$ in the stack. That is, for $i = 2, \ldots, k$, $v_i \in$ Successors(v_{i-1}).

RFRONTIER and SFRONTIER store sets of pairs $\langle v, a\rangle$, where v is a vertex and a is a parse action yet to be performed at v. The vertices of these pairs form the active stack tops. `Shift`, `combine`, and `accept` actions are stored in SFRONTIER. `Reduce`, `eager-reduce`, and `pack-check` actions are stored in RFRONTIER. RFRONTIER is a priority queue. Items on RFRONTIER are ordered by the depth of the vertices that their actions create in the stack. Actions that create vertices deeper in the stack have a higher priority and are run earlier.

The sets CURRENT-V and EAGER-V are used to keep track of possible packing opportunities. CURRENT-V is a set of all vertices whose associated forest node contains at least one derivation that was completed at the current input word. EAGER-V is a set of all vertices created by eager reduction whose associated forest node as yet contains no complete derivation.

ω denotes the current input symbol.

Main Loop

- Add a terminator symbol $ to the end of the input string z
- $\omega \leftarrow$ The first symbol of z
- $v_0 \leftarrow \langle s_0, \text{NIL}, 0, \{\}, \{\}\rangle$ · Initialise the stack
- Call **Schedule**(v_0, ω)
- CURRENT-V $\leftarrow \{v_0\}$
- EAGER-V $\leftarrow \{\}$
- Loop · Perform reductions followed by shifts until acceptance or rejection.
  - Call **Reduce**()
  - Call **Shift**()
  - If SFRONTIER = $\{\langle v, \texttt{accept}\rangle\}$ then halt and return Node(v)
  - If SFRONTIER = $\{\}$ then halt and signal an error.

Reduce()

Perform all outstanding reductions by calling the subroutine appropriate to each reduce action. Actions with the same priority are processed in the following order: completing reduces, full reduces, packing checks, then eager reduces.

- While RFRONTIER $\neq \{\}$
  - $\mathcal{B} \leftarrow$ The set of all items in RFRONTIER with the highest priority
  - If $\exists x \in \mathcal{B}$ s.t. $x = \langle v,$"`reduce` $X \rightarrow \alpha$"$\rangle$
    - $\mathcal{C} \leftarrow \{c \in \textsf{Combine-Ptrs}(v) \mid \textsf{Rule}(c) = X \rightarrow \alpha\}$
    - If $\mathcal{C} \neq \{\}$
      - Call **Completing-Reduce**$(v, \mathcal{C})$

      Else
      - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length $|\alpha|$ starting at v in the stack$\}$
      - $\forall p \in \mathcal{P}$, call **Full-Reduce**$(p, X \rightarrow \alpha)$

    Else If $\exists x \in \mathcal{B}$ s.t. $x = \langle v,$ "`pack-check`"$\rangle$
    - Call **Packing-Check**$(\{v \mid \langle v,$"`pack-check`"$\rangle \in \mathcal{B}\})$

    Else If $\exists x \in \mathcal{B}$ s.t. $x = \langle v,$"`eager-reduce` r, k"$\rangle$
    - $\mathcal{P} \leftarrow \{p \mid p$ is a path of length k starting at v in the stack$\}$
    - $\forall p \in \mathcal{P}$, call **Eager-Reduce**$(p, r)$

- Each element of $\mathcal{C}$ corresponds to a previous eager reduction that is now complete.

Completing-Reduce$(v, \mathcal{C})$

Perform a completing reduction at vertex v. As this reduction covers the same ground as a previous eager reduction, there is no new parse structure to create, and all that needs to be done is schedule any actions at the vertices associated with the result of the completing reduction.

- $\forall c \in \mathcal{C}$
  - Status(Deriv(c)) $\leftarrow$ `complete`
  - Remove c from Combine-Ptrs(v)
  - $\forall v' \in$ Vertices(c)
    - Remove v' from EAGER-V
    - Add v' to CURRENT-V
    - Add $\langle v',$"`pack-check`"$\rangle$ to RFRONTIER with priority Depth(v')

- $\mathcal{C}$ is a set of combine pointers that point to the vertices created earlier by an eager reduction that is now being completed.
- Check that the packing of Node(v') is still correct.

Full-Reduce$(v_1 \ldots v_k, X \rightarrow \alpha)$

Non-eagerly reduce by the rule $X \rightarrow \alpha$, creating a derivation of X whose children are the nodes of vertices $v_1 \ldots v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by this reduction.

· $d \leftarrow \langle(\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1)), \texttt{complete}\rangle$	
· $\Pi \leftarrow$ A partition of $\mathsf{Successors}(v_k)$ by goto value on symbol X	· v belongs to the set $\pi_s \in \Pi$ if and only if "$\texttt{goto}\ s$" $\in$ ACTION[$\mathsf{State}(v)$,X]. See figure 1(a).
· If $\exists v \in$ CURRENT-V s.t. $\mathsf{Successors}(v) \in \Pi \wedge \mathsf{Nonterm}(\mathsf{Node}(v)) = X$	· Pack using an existing vertex in CURRENT-V whose nonterminal and left context match the current reduction.
· Add d to $\mathsf{Derivs}(\mathsf{Node}(v))$	
Else	
· $n' \leftarrow \langle X, \{d\}\rangle$	· Create a new forest node.
· $\forall \pi_s \in \Pi$	· Create new vertices containing this node, one for each element of Π.
· $l \leftarrow \mathrm{Max}(\{\mathsf{Depth}(v) \mid v \in \pi_s\}) + 1$	
· $v' \leftarrow \langle s, n', l, \{\}, \pi_s\rangle$	
· Add v' to CURRENT-V	
· $\forall v \in \pi_s$	· Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have $\mathsf{Node}(v)$ as their rightmost element.
· $\forall a \in$ ACTION[$\mathsf{State}(v)$, ω] s.t. a = "$\texttt{combine}\ r$"	
· Call **Combine**(v, v', r)	
· Call **Schedule**(v', ω)	

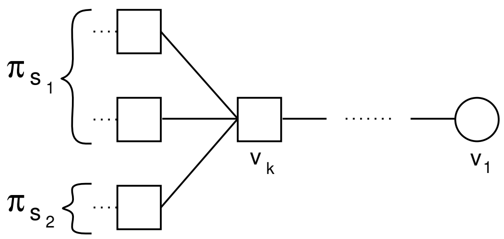

Figure 1(a): Stack before full reduction

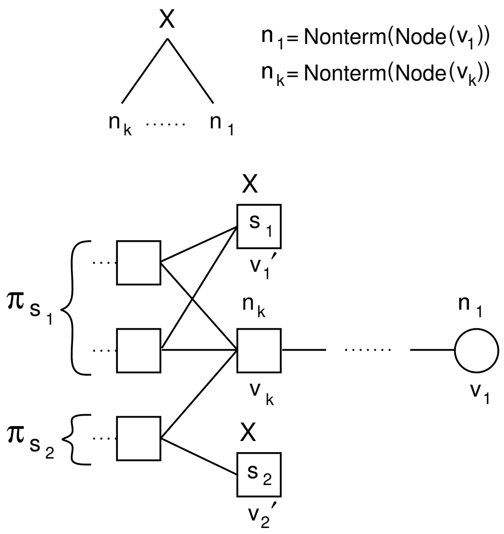

Figure 1(b): Stack after full reduction

Eager-Reduce$(v_1 \ldots v_k, X \rightarrow \alpha)$

Eagerly reduce by the rule $X \rightarrow \alpha$, creating a new incomplete derivation of X whose children are the nodes of vertices $v_1 \ldots v_k$. Combine this new node into other partial derivations created previously by eager reduction, and schedule any further actions triggered by reduction.

· $d \leftarrow \langle(\mathsf{Node}(v_k), \ldots, \mathsf{Node}(v_1)), \texttt{incomplete}\rangle$	· The new derivation is incomplete because it is created by an eager reduction.
· $\Pi \leftarrow$ A partition of Successors(v_k) by goto value on symbol X	· v belongs to the set $\pi_s \in \Pi$ if and only if "$\texttt{goto}\ s$" $\in$ ACTION[State(v),X]. See figure 2(a).
· $c \leftarrow \langle X \rightarrow \alpha, d, \{\}\rangle$	· Create a new combine pointer.

· $\forall v \in$ EAGER-V
  · If Successors$(v) \in \Pi \wedge$ Nonterm(Node(v)) = X
    · Add v to Vertices(c)
	· Collect the set of existing eager vertices whose nonterminal and left context match the current reduction.

· If Vertices$(c) \neq \{\}$ then	· Pack using an existing vertex created by a previous eager reduction.
· $v \leftarrow$ An arbitrary element of Vertices(c) · Add d to Derivs(Node(v))	· All elements of Vertices(c) have the same forest node.
Else	
· $n' \leftarrow \langle X, \{d\}\rangle$	· Create a new forest node.

· $\forall \pi_s \in \Pi$
    · $l \leftarrow$ Max$(\{$Depth$(v) \mid v \in \pi_s\}) + 1$
    · $v' \leftarrow \langle s, n', l, \{\}, \pi_s\rangle$
    · Add v' to EAGER-V
    · Add v' to Vertices(c)
	· Create new vertices containing this new node, one for each element of Π.

· $\forall v \in \pi_s$
      · $\forall a \in$ ACTION[State(v), ω] s.t. a = "$\texttt{combine}\ r$"
        · Call **Combine**(v, v', r)
	· Combine the newly created n' into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.

· Call **Schedule**$(v'$,*EAG*$)$	· Use *EAG* because appropriate lookahead symbols are not available.
· Add c to Combine-Ptrs(v_1)	

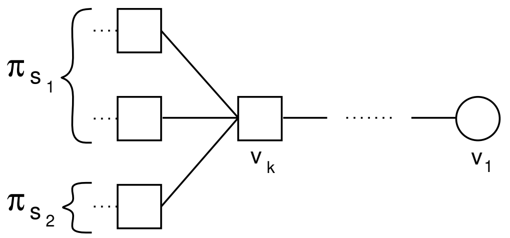

Figure 2(a): Stack before eager reduction

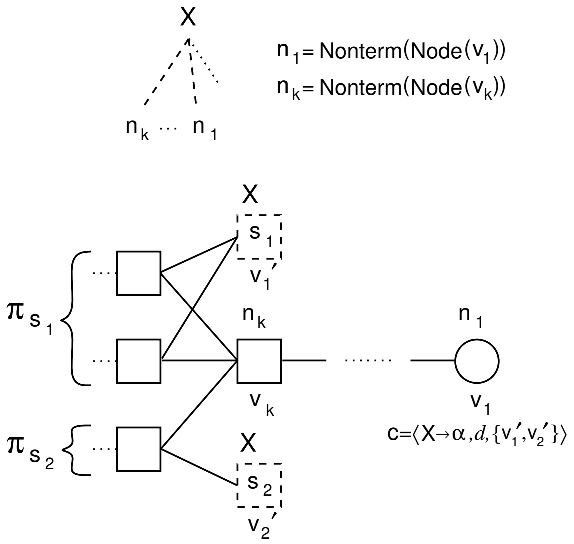

Figure 2(b): Stack after eager reduction

Combine(v_f, v_t, r)

Combine Node(v_t) into partial derivations created by eager reduction whose rightmost element is currently Node(v_f). These derivations are identified by the combine pointers associated with v_f.

- $\forall c \in$ Combine-Ptrs(v_f) s.t. Rule(c) = r
  - Add Node(v_t) to the end of Children(Deriv(c))
  - Remove c from Combine-Ptrs(v_f)
  - Add c to Combine-Ptrs(v_t)

- Move the combine pointers forward to v_t so that further combines (or completing reductions) can be performed there.

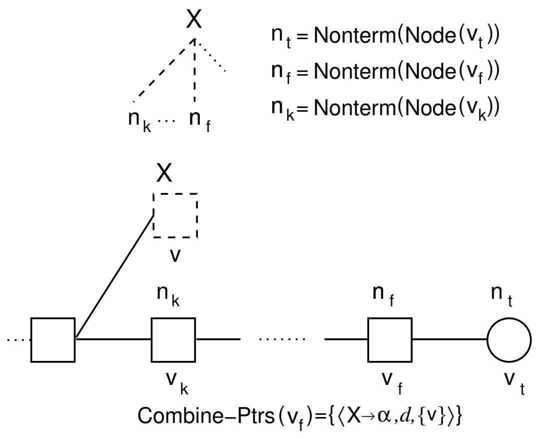

Figure 3(a): Stack before combine

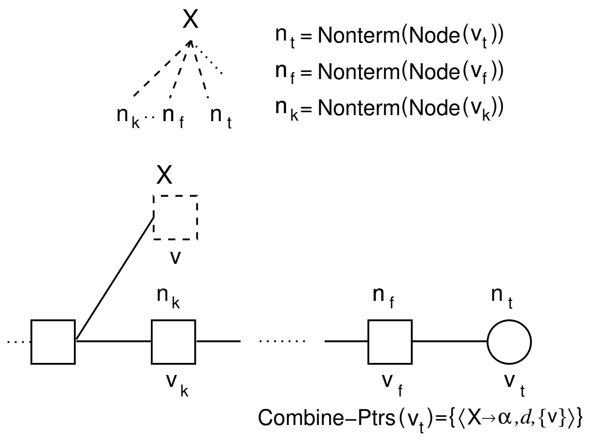

Figure 3(b): Stack after combine

Packing-Check($\mathcal{L}$)

Check the provisional packings of nodes in the parse forest. If there are both complete and incomplete derivations packed together in the same node, then the packing was incorrect and the incomplete and complete derivations must be separated and unpacked into two distinct forest nodes. The complete derivations were created by completing reductions at the current input word, so schedule any actions these completing reductions trigger at the vertices whose associated forest node contain the complete derivations.

· $\Pi \leftarrow$ A partition of $\mathcal{L}$ according to associated forest node	· Each v belongs to the set $\pi_n \in \Pi$ if and only if Node$(v) = n$.
· $\forall \pi_n \in \Pi$	
· $\mathcal{C} \leftarrow \{d \in$ Derivs(n) | Status(d) = `complete` $\}$	
· $\mathcal{I} \leftarrow \{d \in$ Derivs(n) | Status(d) = `incomplete` $\}$	
· If $\mathcal{C} \neq \{\}$ then	
· If $\mathcal{I} \neq \{\}$ then	· Forest node n contains both complete and incomplete derivations, so it is necessary to unpack.
· $n' \leftarrow \langle$Nonterm$(n), \mathcal{C}\rangle$	· Remove the complete derivations from n and put them into a new node n'. See figure 4(b).
· Derivs$(n) \leftarrow \mathcal{I}$	
· $\forall v \in \pi_n$	
· Remove v from CURRENT-V	
· Add v to EAGER-V	
· $v' \leftarrow \langle$State(v),n',Depth(v),$\{\}$,Successors$(v)\rangle$	· Create new vertices associated with the newly created node.
· Add v' to CURRENT-V	
· Call **Schedule**(v', ω)	· Node(v') contains the complete derivations, so schedule futher actions at v'
Else	
· $\forall v \in \pi_n$	· All $v \in \pi_n$ have forest node n, and all derivations of n are complete; therfore schedule further actions at these vertices.
· Call **Schedule**(v, ω)	

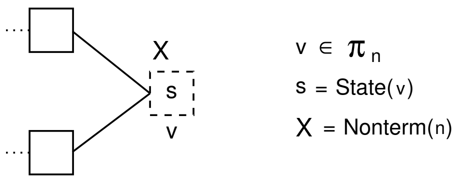

Figure 4(a): Stack before unpacking

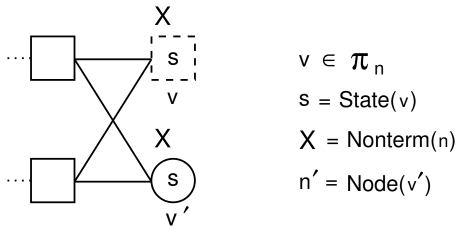

Figure 4(b): Stack after unpacking

Shift()

Shift the next terminal symbol onto all the stack tops and create a new node for it in the parse forest. Combine this new node into other partial derivations created previously by eager reduction, and schedule any actions triggered by the shift.

- $n \leftarrow \langle \omega, \{\} \rangle$ — · Create a new node for the shifted input symbol.
- $\omega \leftarrow$ The next symbol of the input string
- CURRENT-V $\leftarrow \{\}$
- $\mathcal{S} \leftarrow \{\langle v, a \rangle \in$ SFRONTIER $\mid a =$ "`shift` s"$\}$ — · $\mathcal{S}$ is the set of all shift actions to perform.
- SFRONTIER $\leftarrow$ SFRONTIER $- \mathcal{S}$
- $\Pi \leftarrow$ A partition of $\mathcal{S}$ according to goto state of the `shift` actions — · Each $\pi_s \in \Pi$ consists of elements of the form $\langle v,$ "`shift` s" $\rangle$.
- $\forall \pi_s \in \Pi$ — · Create a new vertex for each member of Π.
  - $\mathcal{V} \leftarrow \{v \mid \langle v, a \rangle \in \pi_s\}$
  - $l \leftarrow$ Max($\{$Depth$(v) \mid v \in \mathcal{V}\}) + 1$
  - $v_s \leftarrow \langle s, n, l, \{\}, \mathcal{V} \rangle$
  - Add v_s to CURRENT-V
  - $\forall x \in$ SFRONTIER s.t. $x = \langle v,$"`combine` r"$\rangle \wedge v \in \mathcal{V}$ — · Combine the newly created n into previous eager reductions by rule r whose corresponding partial derivations have Node(v) as their rightmost element.
    - Remove x from SFRONTIER
    - Call **Combine**(v, v_s, r)
  - Call **Schedule**(v_s, ω)

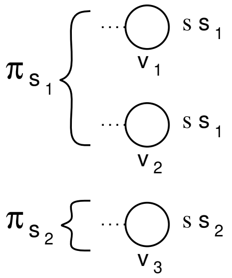

Figure 5(a): Stack tops with outstanding shift actions

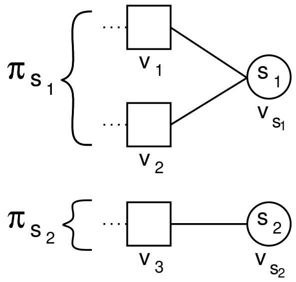

Figure 5(b): Stack after shifting

Schedule(v, X)

Add to RFRONTIER and SFRONTIER all possible actions to be performed at vertex v.

- $\forall a \in$ ACTION[State$(v), X$] s.t. $a =$ "`eager-reduce` r, k" $\wedge$ $\neg\exists c \in$ Combine-Ptrs(v) s.t. Rule$(c) = r$ — · Filter out reductions that would repeat an eager reduction carried out earlier.
  - $p \leftarrow$ Depth$(v) - k + 1$
  - Add $\langle v, a \rangle$ to RFRONTIER with priority p
- $\forall a \in$ ACTION[State$(v), X$] s.t. $a =$ "`reduce` $X \rightarrow \alpha$"
  - $p \leftarrow$ Depth$(v) - |\alpha| + 1$
  - Add $\langle v, a \rangle$ to RFRONTIER with priority p
- $\forall a \in$ ACTION[State$(v), X$] s.t. a is a `shift`, `combine`, or `accept` action
  - Add $\langle v, a \rangle$ to SFRONTIER

□