RECURSIVE ASCENT-DESCENT PARSING

R. Nigel Horspool
Department of Computer Science, University of Victoria,
P.O. Box 1700, Victoria, B.C.
Canada V8W 2Y2

11 June 1991

Abstract

Generalized left-corner parsing was originally presented as a technique for generating a parser for the SLR(1) class of grammars but with far fewer states than the SLR(1) parser. This paper modifies and extends the formulation of left-corner parsers so that it is possible to apply the technique to the LALR(1) and LR(1) classes of grammars. It is further shown that left-corner parsers can be converted into directly executed code in a manner that subsumes the parsing methods known as recursive descent and recursive ascent – hence the name *recursive ascent-descent*. The directly executed form has the advantage that it allows a compiler writer to insert semantic code into the parser incrementally, without having to re-execute the parser generator.

Index Terms compilers, parsers, left-corner parsing, recursive descent, recursive ascent.

1 Introduction

A parsing technique called generalized left-corner parsing, GLC, was introduced by Demers[7]. The technique may be viewed as a hybrid between the top-down and bottom-up approaches to parsing. Very informally, a bottom-up parsing approach is used to recognize the input until it becomes unambiguous as to which grammatical production rule is being matched. A top-down approach is then used to match the remainder of the production rule's right-hand side.

As presented by Demers, the test used to resolve conflicts between alternative parsing choices is the same as that employed by the SLR(1) parsing method[8]. Consequently, the construction algorithm that was given for creating a left-corner parser was closely related to the SLR(1) parser construction algorithm, and the class of acceptable grammars is the SLR(1) class. The parsing method was dubbed SGLC(1) to indicate that it was a SLR(1)-based version of generalized left-corner parsing. Although it was stated that the technique is easily generalized to the LR(k) grammar classes, no details were provided. The main advantage given for SGLC(1) parsing was that the parser would have fewer recognizer states than the corresponding SLR(1) parser, and implementations should thus require less memory.

Nowadays, the SLR(1) parsing method has been almost totally supplanted by the LALR(1) and LR(1) parsing methods because they accept larger classes of grammars. The LALR(1) method is clearly superior because it accepts a larger grammar class without increasing the number of states over the SLR(1) method. In this paper, we will show how the left-corner parsing approach can be extended to accommodate the LALR(1) and LR(1) grammar classes. This extension requires a modification to the original formulation of left-corner parsing to be made. Using the prefix 'X' (instead of 'G') to denote *extended*, we call the LALR(1)-based version of the method LAXLC(1) and the LR(1)-based version XLC(1).

The benefits of this work are two-fold. First, we retain the original advantage of left-corner parsing in that the parser will almost always have many fewer states than the equivalent LALR(1) or LR(1) parser. Secondly, we will show that the left-corner parser can be converted into directly executable code in a manner that subsumes the recursive-descent [1] and recursive-ascent [14, 15, 3] methods. Any user who demands the ability to insert semantic code directly into the generated parser will find the directly executable form of a left-corner parser advantageous.

In the remainder of this paper, we review left-corner parsing and then we describe the extended left-corner parsing method. Subsequently, we review the recursive-descent and recursive-ascent methods and then show how our left-corner parser can be converted into a directly executable version that we call a recursive-ascent-descent parser. Finally, we conclude with a report on some experience with an implementation of a generator for LAXLC(1) parsers (LALR(1)-based extended generalized left-corner parsers).

We assume that the reader has moderate familiarity with the LALR(1) and LR(1) parsing methods. Introductions to this material may be found in many texts, including [1], [6] and [9].

2 Notational Conventions

Throughout the paper, some standard notational conventions are used. A (context-free) grammar G is a four-tuple $< V_N, V_T, P, S >$ where V_N is a finite set of non-terminal symbols, V_T is a finite set of terminal symbols ($V_T \cap V_N = \Phi$), $S \in V_N$ is the start symbol, and P is a list of production rules of the form $A \rightarrow \alpha$ where $A \in V_N$ and $\alpha \in \{V_N \cup V_T\}^*$.

We use additional conventions that

A,B,C ... denote non-terminal symbols,
a,b,c ... denote terminal symbols,
X,Y,Z ... denote symbols in $V_N \cup V_T$,
v,w,x ... denote strings in ${V_T}^*$,
α, β, γ, ... denote strings in $(V_N \cup V_T)^*$,
ϵ denotes the empty string,
$\dashv$ denotes an end-of-input marker

We write $\alpha A \gamma \Rightarrow \alpha\beta\gamma$ if $A \rightarrow \beta \in$ P. The relation $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$. The notation L(G) represents the language generated by G and is defined as follows.

$$L(G) \quad = \quad \{ \alpha \mid S \Rightarrow^* \alpha \wedge \alpha \in {V_T}^* \}$$

The function first_k is defined as follows.

$$\text{first}_k(\alpha) \quad = \quad \{\ v \mid \exists x :\ \alpha \Rightarrow^* vx \ \wedge\ |v| = k\ \}$$

For convenience, we also use a two-argument variation of first_k, where the second argument is a set of suffixes to be appended to the first argument. It is defined as follows.

$$\text{first}_k(\alpha, \sigma) \quad = \quad \bigcup_{w \in \sigma} \text{first}_k(\alpha w)$$

We assume throughout that G is a *reduced* grammar. That is, for every production rule p there must exist a sentence in L(G) whose derivation uses p.

LR(0) items are production rules in which a position in the right-hand side is indicated by a dot marker. We use the notation $[\, A \rightarrow \alpha \bullet \beta\,]$, where $A \rightarrow \alpha\beta \in$ P, for an item.

3 Overview of SGLC(1) Left-Corner Parsing

As formulated by Demers, the construction algorithm for the left-corner parser must initially determine a *recognition point* for each production rule in the grammar. This is the point in the rule's right-hand side which has been reached before the parser has unambiguously determined that it is this particular rule that is being matched. Furthermore, (and this will be a point of difference between our approach and Demers' approach) it is necessary that it remain unambiguous as to which rule is being used while remaining symbols on the right-hand side are matched. Every rule must have a recognition point, even if this point occurs at the very end of the rule's right-hand side.[1] The positions of recognition points are necessarily a property of the parsing method. For example, we can easily construct grammars for which the SLR(2) parsing method would know one symbol sooner than the SLR(1) method which rule is being recognized.

Determination of recognition points is not a trivial matter. Demers only gave a practical algorithm that works when the parsing method is SLR(1) and when the grammar has no left-recursive rules. For SLR(1) grammars that do have left-recursive rules, it is implicitly suggested that a trial-and-error approach should be used to find the recognition points. (We will later introduce a generalization to the notion of recognition point and it makes the problem of determining recognition points equivalent to another problem for which a general algorithm has been published.) We will omit a description of the Demers algorithm and simply give an example grammar in which the recognition point of rule i is marked by the symbol $\widehat{i}$.

1.	A	$\rightarrow$	a	B	b	$\widehat{1}$	C
2.	B	$\rightarrow$	B	b	$\widehat{2}$		
3.	B	$\rightarrow$	$\widehat{3}$	b			
4.	C	$\rightarrow$	C	$\widehat{4}$	c		
5.	C	$\rightarrow$	$\widehat{5}$	c			

(Grammar G_1)

[1] If we could reach the end of a right-hand side without unambiguously knowing which rule is involved, we would not be able to reduce by that rule at that moment, and hence we could not construct a canonical parser.

State	T_Table				N_Table		
	a	b	c	$\dashv$	A	B	C
$Q_0 = Q_A$	s2				s8		
$Q_1 = Q_C$			$\widehat{5}$				s5
Q_2		$\widehat{3}$				s3	
Q_3		s4					
Q_4		$\widehat{2}$	$\widehat{1}$				
Q_5			$\widehat{4}$	P2			
$Q_6 = Q_b$		s8					
$Q_7 = Q_c$			s8				
Q_8	P2	P2	P2				

Rule	Pops	LHS	RHS Suffix
1	3	A	Q_C
2	2	B	
3	0	B	Q_b
4	1	C	Q_c
5	0	C	Q_c

Figure 1: Left-Corner Parse Tables for Grammar G_1

The grammar has been contrived to illustrate a point that is made later when we describe our improvements to the parsing method.

As with other LR parsers, the SGLC parser for this grammar uses a stack that can hold state numbers. For this parser, five of the states are associated with grammar symbols. A state with a name like Q_a indicates that the state is associated with symbol a, etc. The symbol $\dashv$ has been used to represent the end-of-input.

The parse tables for this grammar are shown in Figure 1. The table entries have the following meanings and cause the following actions to occur. An entry of the form sk means 'shift to state k'; it causes state number k to be pushed onto the stack and a new terminal symbol to be read. An entry of the form $\widehat{\imath}$ means 'announce rule i' and the associated action has three parts to it. Let rule i have the form $A \rightarrow \alpha\,\widehat{\imath}\,\beta$. Then the first part of the action is to pop $|\alpha|$ items off the stack, exposing *top* as the state on top of the stack. The second part is to push the state in the $N\_Table[top, A]$ entry onto the stack. The third part is to push the states that correspond to the individual symbols of β onto the stack in reverse order. (The table indexed by the rule number in Figure 1 could be used to implement the *announce* action; it gives the number of entries to pop, the left-hand side symbol of the rule, and the sequence of states to be pushed.) An entry of the form Pk means 'pop k entries off the stack'; for a SGLC parser, k is always equal to 2. (We introduce the parameter to the *pop* action now so that the SGLC(1) table entries have the same form as for the LAXLC(1) and XLC(1) parsers.) As usual, a blank entry in the *T_TABLE* indicates a syntax error. (Blank entries in the *N_TABLE* represent "don't care" entries.)

The parse begins by pushing the state associated with the start symbol of the grammar (Q_A in this case), onto the stack and reading the first terminal symbol from the input. The parser repeatedly applies the action $T\_Table[top, x]$ where *top* is the top state on the stack and x is the current input symbol. The parser halts if the stack becomes empty and it would then report success only if the input has been exhausted (i.e. if the current symbol is the end-of-file marker $\dashv$).

4 Extended Generalized Left-Corner Parsing

4.1 Free Positions and Rule Recognition Points

The notion of a rule recognition point can be generalized and put on a more formal basis as follows. Given a grammar G that includes a production p

$$\texttt{A} \quad \rightarrow \quad \alpha \ \beta$$

(where α and β cannot both be empty) then a particular canonical parsing method M can be said to capable of determining that this rule is being matched after having read the symbols of α if the following two conditions hold. First, M must be capable of recognizing exactly the language L that is generated by G. Second, M must be able to recognize the same language L using the modified grammar, G$'$, which is identical to G except that the production p is replaced by the two rules

$$\begin{array}{lcl} \texttt{A} & \rightarrow & \alpha \ \texttt{Z} \ \beta \\ \texttt{Z} & \rightarrow & \epsilon \end{array}$$

where Z is a new (and therefore unique) non-terminal symbol.

The above definition provides an inefficient but constructive algorithm for testing each position in the right-hand side of every production to check if it is a recognition point. There are two remarks to make.

First, our generalization of recognition point is identical to the notion of a *free position* used by Purdom and Brown[13]. We will therefore use the name 'free position' throughout the remainder of this paper. The importance of a free position is that it represents a point in the parse where semantic action code can be freely inserted. A common technique for attaching semantic code is to insert a new non-terminal symbol at the point in the grammar where the action is to be performed; to define the new non-terminal by a null production; and to execute the desired semantic action code whenever the parser reduces by that null production. Our definition for free position corresponds to that technique.

Second, it is implicitly required by Demers' formulation of left-corner parsing that *every* position in a rule that follows a recognition point must also be a possible recognition point. The leftmost recognition point in each rule should be used in order to minimize the number of parser states. If we generalize from recognition point to free position, we do not necessarily find that all positions following a free position are free.

The first rule in the example grammar given previously illustrates the situation. The entire grammar is reproduced below using the symbol $\diamond$ to indicate a free position. For this grammar, it does not matter whether the parsing method is SLR(1), LALR(1) or LR(1), the same set of free positions will result.

$$\begin{array}{lllll} 1. & \texttt{A} & \rightarrow & \diamond \ \texttt{a} \ \diamond \ \texttt{B} \ \texttt{b} \ \diamond \ \texttt{C} \ \diamond \\ 2. & \texttt{B} & \rightarrow & \texttt{B} \ \texttt{b} \ \diamond \\ 3. & \texttt{B} & \rightarrow & \diamond \ \texttt{b} \ \diamond \\ 4. & \texttt{C} & \rightarrow & \texttt{C} \ \diamond \ \texttt{c} \ \diamond \end{array}$$

5. C → ◇ c ◇

(The reader is invited to test that the free positions are marked correctly by supplying the grammar to a parser generator and inserting semantic actions at various points.) In contrast to the left-corner parsing method of Demers, we can always use the leftmost free position in a rule as the recognition point.

4.2 The Construction Algorithm for LAXLC(1) Parsers

We now give the parser construction algorithm for the LALR(1)-based version of our extended left-corner parsing method. As explained later, the construction algorithm requires only minor changes to become SLR(1)-based or LR(1)-based.

Given a grammar G, a preliminary step in constructing the extended left-corner parser is to determine the free positions in the production rules of G. This requires construction of the LR(0) sets of items for G with conflicts resolved by standard LALR(1) analysis of context sets. If any of the conflicts cannot be resolved, we reject G and do not proceed any further. The Purdom and Brown algorithm[13] may now be applied to the item sets to discover which positions are free.

In each rule, one free position must be selected as the recognition point of that rule. Normally, we would choose the leftmost free position in each rule as this would usually create a parser with the fewest states. However, the construction method works correctly if *any* free position is chosen. Following Demers, we use the marker $\widehat{\imath}$ to indicate the recognition point of rule i. Thus, a LR(0) item written as $[\,A \rightarrow \alpha \bullet X\beta\,]$ implies that the dot marker is *not* at the recognition point. When the dot is at the recognition point, we write the item as $[\,A \rightarrow \alpha \bullet \widehat{\imath}X\beta\,]$.

Once the recognition points have been selected, the grammar is inspected to create a number of basis items. For every production rule $A \rightarrow \alpha\widehat{\imath}\beta$, we decompose β into zero or more non-empty substrings that do not contain any free positions. Let $\beta = \beta_1 \diamond \beta_2 \diamond ... \beta_k$, where the $\diamond$ symbol is used to mark each free position. Then for each β_j, we create a basis item set

$$\{\,[\,S_{\beta_j} \rightarrow \vdash \bullet \beta_j\,]\,\}$$

for an entry state named Q_{β_j}. We use $\vdash$ as a *start of input* symbol, and we use the new symbols of the form S_α to represent subgoals, where $\vdash \notin V_T$ and $S_\alpha \notin V_N$. The $\vdash$ symbol is used in items to enforce the property that the dot marker never appears immediately after the production arrow in a basis item. In addition, the basis item set

$$\{\,[\,S_S \rightarrow \vdash \bullet S\,]\,\}$$

for a state named Q_S is created, where S is the goal symbol of the grammar. Each distinct basis item set is used to derive a separate entry state for the recognizer. Q_S is the start state for the recognizer; the other entry states are used for subgoal recognition when the parser is operating in a top-down mode.

For the purposes of defining the closure and goto functions below, we assume that a rule with a subgoal symbol S_α on the left-hand side contains no free positions and contains no recognition point marker.

A closure function cl is applied to a set of items to create the complete set of items that comprises a recognizer state. If I is a set of items, then $cl(I)$ is the smallest set of items such that $I \subseteq cl(I)$ and if $[\, A \rightarrow \alpha \bullet B\beta \,] \in I$, then $[\, B \rightarrow \bullet \gamma \,] \in I$ for each rule $B \rightarrow \gamma$ in G. This is the same closure process as for construction of a LR(0) recognizer with the exception that if the dot should be at the recognition point of the rule used in an item, no closure items would be generated from that item.

Additional states, reached on terminal or non-terminal transitions out of an entry state, are constructed by the *goto* function. If I is a set of items and X is a grammar symbol, $goto(I, X)$ is defined as the closure of the set of items $[\, A \rightarrow \alpha X \bullet \beta \,]$ such that $[\, A \rightarrow \alpha \bullet X\beta \,] \in I$. Again, this is identical to the usual LR(0) approach except that an item in I that has its dot marker at the rule's recognition point cannot contribute a basis item to a goto state.

The complete collection of recognizer states is obtained by applying the closure function to each distinct set of basis items created on the initial pass through the grammar. This creates one or more sets of items corresponding to entry states of the recognizer. The goto function is now repeatedly applied to every distinct set of items for every grammar symbol, until no new distinct sets of items can be created.

If every position to the right of the recognition point in each rule is free, then our algorithm for constructing the sets of items is essentially the same as Demers'. A difference only arises if there is a gap in the sequence of free positions, as grammar G_1 exemplifies.

Once all the sets of items have been created, the final task is to associate context sets with items. The association may be defined in terms of a mapping function: $context(Q, i) \in {V_T}^*$ is the context associated with item i in state Q. We will describe an approach equivalent to that used in construction of LALR(1) parsers. The context sets are needed for resolving shift-reduce and reduce-reduce conflicts.

The basis item for the goal symbol is given a context set of $\{\dashv\}$. In other words, we define

$$context(Q_S, [\, S_S \rightarrow \vdash \bullet S \,]) \;=\; \{\dashv\}$$

All other context sets must satisfy the following two inclusion requirements.

$$\forall Q: \; context(Q, [\, A \rightarrow \alpha \bullet B\beta \,]) \;\subseteq\; context(Q, [\, B \rightarrow \bullet \gamma \,]), \quad B \rightarrow \gamma \in P$$

$$\forall Q: \; context(Q, [\, A \rightarrow \alpha \bullet X\beta \,]) \;\subseteq\; context(goto(Q, X), [\, A \rightarrow \alpha X \bullet \beta \,])$$

There is one additional requirement for the context sets of subgoal entry states. If

$$context(Q, [\, A \rightarrow \alpha \bullet \widehat{\imath}\beta_1 \diamond \beta_2 \diamond \ldots \beta_k \,]) \;=\; \sigma$$

then

$$\forall j (1 \leq j \leq k): \; \sigma \;\subseteq\; context(Q_{\beta_j}, [\, S_{\beta_j} \rightarrow \vdash \bullet \beta_j \,])$$

Clearly, a solution to these set inclusion requirements must exist – we could simply choose every context set to be V_T. However, to resolve as many potential parsing conflicts as possible, we desire the smallest context sets that satisfy the inclusion requirements. An iterative procedure

may be used to compute these sets. However, we note that, analogously to LALR(1) parser construction, more efficient ways of computing context sets than blind iteration exist.

Once the context sets have been determined, we can create the entries of the two parsing tables, *T_TABLE* and *N_TABLE*. In the following, we use Q (and Q') to refer both to a set of items and to the corresponding recognizer state. It should be clear by context which meaning is intended.

$T\_TABLE[Q,a]$ = *announce rule i* if Q contains an item $[A \to \alpha \bullet \widehat{\imath}\beta]$ with an associated context set of σ, and $a \in \text{first}_1(\beta, \sigma)$.

$T\_TABLE[Q,a]$ = *shift to state* Q' if Q contains an item $[A \to \alpha \bullet a\beta]$ and $goto(Q,a) = Q'$.

$T\_TABLE[Q,a]$ = *pop* k if Q contains an item $[S_\alpha \to \vdash \alpha \bullet\ ]$ with associated context set σ, $a \in \sigma$, and $|\vdash \alpha| = k$.

$N\_TABLE[Q,B]$ = *shift to state* Q' if Q contains an item $[A \to \alpha \bullet B\beta]$ and $goto(Q,B) = Q'$.

If the above rules supply at most one action for each table entry in *T_TABLE* and *N_TABLE*, the parser is a deterministic recognizer for the language L(G). If more than one action should be provided in some position, the parser would still be a recognizer for L(G), but it would be non-deterministic. (A non-deterministic choice would have to be made between the multiple entries when the parser is executed.) It is guaranteed that a deterministic parser will be created for a LALR(1) grammar, as proved in the appendix.

One more point about the LAXLC(1) construction algorithm should be noted. If the recognition point is chosen to be at the extreme right of each rule (a position which is necessarily free for a LALR(1) grammar), our construction algorithm is identical to the LALR(1) construction algorithm. (There is actually an insignificant difference in that the start state of the LAXLC(1) recognizer will contain the extra item $[S_S \to \vdash\ \bullet\ S]$, and the recognizer will contain an explicit halt state that has the item set $\{\,[S_S \to \vdash S \bullet\ ]\,\}$.)

The parser tables that are constructed by the LAXLC(1) construction algorithm for grammar G1 are shown in Figure 2. We have taken the liberty of optimizing the tables slightly by introducing a new kind of action written as '*Pk'. This action is to be read as 'read and pop'; it causes a new input symbol to be read and k entries to be popped from the stack. Its effect is identical to executing a 'shift' action to a state in which the next action is 'pop $k+1$'. [The SGLC(1) tables of Figure 1 contain an equivalent optimization because all the states that should contain just a single 'pop two' entry and nothing else have been merged into a single state (state Q_8).]

A proof that the LAXLC(1) construction algorithm correctly constructs recognizers for the LALR(1) class of grammars is given in the appendix.

4.3 LR(1)-Based Construction Algorithm

The LR(1)-based version of the extended, generalized left-corner parser construction algorithm is easy to formulate. The initial difference is that we must use LR(1) items instead of LR(0) when

State	T_Table				N_Table		
	a	b	c	$\dashv$	A	B	C
$Q_0 = Q_A$	$\widehat{1}$						
$Q_1 = Q_{Bb}$		$\widehat{3}$				s6	
$Q_2 = Q_C$			$\widehat{5}$				s8
$Q_3 = Q_a$	*P1						
$Q_4 = Q_b$		*P1					
$Q_5 = Q_c$			*P1				
Q_6		s7					
Q_7		$\widehat{2}$	P3				
Q_8			$\widehat{4}$	P2			

Rule	Pops	LHS	RHS Suffix
1	0	A	$Q_a\ Q_{Bb}\ Q_C$
2	2	B	
3	0	B	Q_b
4	1	C	Q_c
5	0	C	Q_c

Figure 2: LAXGLC(1) Parser Tables for Grammar G1

constructing states. That is, an item will have the form $[\, A \rightarrow \alpha \bullet \beta \,;\, \sigma \,]$ where $\sigma \in {V_T}^*$ is a set of context symbols for the item. This means that two sets of items that differ only in their context sets would give rise to two distinct states in the LR(1)-based recognizer but to a single state in the corresponding LALR(1)-based recognizer. The increase in the size of the grammar class that can be accommodated is therefore offset by an increase in the number of states and hence in the size of the parser. This trade-off is not usually considered to be worthwhile, and LALR(1) parser generators are commonly preferred to LR(1) parser generators.

The XLC(1) construction algorithm is a straightforward generalization of the LAXLC(1) construction algorithm. The relationship between the two methods is similar to that between LALR(1) and LR(1). For this reason, we omit details of the algorithm.

4.4 Handling LL(1) Grammars

The left-corner parsing methods are a combination of bottom-up parsing and top-down parsing. The XLC(1) and LAXLC(1) methods subsume the standard bottom-up parsing methods, LR(1) and LALR(1), respectively. They accept the same grammar classes and they generate the same parser tables when the recognition point of each production is placed at the extreme right-hand end.

The standard top-down parsing method used in compiler construction is LL(1). A parser for a LL(1) grammar can have a table-driven implementation or, very commonly, it is implemented using the method known as *recursive descent*. Recursive descent parsers are easy to construct manually from LL(1) grammars and have been used in many compiler implementations. It is desirable, from the point of view of completeness, that the XLC(1) and LAXLC(1) construction algorithms should be able to accept LL(1) grammars.

It has often been claimed that every LL(1) grammar is a SLR(1) grammar (and hence, also, a LALR(1) grammar). Unfortunately, this is not quite true and counterexamples exist[5]. It is

particularly unfortunate because a LL(1) grammar has the property that every position in every rule is free. That is, a semantic action may be inserted at any point in any rule and the grammar still retains its LL(1) property.

Therefore, we suggest that the LAXLC(1) construction algorithm should be augmented with an additional step. If it is discovered that the LALR(1) sets of items (needed in the initial step of determining free positions) have unresolved conflicts, we should then check the grammar to test if it is LL(1). If it is LL(1), we simply mark every position in every rule as free and we must also select the recognition point of each rule as being at the start of its right-hand side. The remaining steps of the LAXLC(1) construction algorithm may then be followed as before. It is straightforward to see that the construction algorithm will create a parser that is equivalent to a LL(1) parser for the grammar.

We note that there is no need to augment the XLC(1) construction algorithm in this way since every LL(1) grammar is a LR(1) grammar.

5 Recursive-Ascent-Descent Parsing

5.1 Recursive-Descent

A LL(1) grammar, G, may be characterized by the following property. For every non-terminal A, each production rule $A \rightarrow \alpha$ in G has a distinct one-symbol predictor set. The predictor set for the rule $A \rightarrow \alpha$ is sometimes written as $DS(A \rightarrow \alpha)$, where DS stands for *director set*, and may be computed as $\text{first}_1(\alpha, \text{follow}(A))$, where

$$\text{follow}(A) \quad = \quad \{\, a \mid S \Rightarrow^* \alpha A a \beta \,\}$$

More detail about LL(1) grammars and efficient algorithms for computing the predictor sets may be found in standard compiler construction texts such as [1, 9].

Possibly the most important aspect of a LL(1) grammar is that it is easily converted into directly executable code that implements a parser for the grammar. The conventional style for this code provides one (recursive) procedure for each non-terminal symbol in the grammar. For a non-terminal A that appears on the left of the rules $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ... $A \rightarrow \alpha_n$, the procedure is named A and has the following structure. In this example (and in subsequent examples), we use

a programming language notation similar to the C language.

```
void A() {
     if (sym ∈ DS(A→α1)) {
          recognize α1;
     } else if (sym ∈ DS(A→α2)) {
          recognize α2;
     } else ...
     ...
     } else if (sym ∈ DS(A→αn)) {
          recognize αn;
     } else
          report a syntax error;
}
```

We assume that the global variable *sym* holds the current terminal symbol.

The code that corresponds to the instruction *recognize* α has a particularly simple structure. If $\alpha = X_1X_2...X_k$ then the code has the following form.

{ *match* X_1; *match* X_2; ... *match* X_k; }

where the instruction *match* X has one of two forms depending on whether X is a terminal or non-terminal symbol. If X is a terminal, then the code for *match* X is as follows.

```
if (sym == X)
     sym = new symbol from input;
else
     report a syntax error;
```

Otherwise, if X is a non-terminal then the code consists of a call to the procedure responsible for matching X, as follows.

```
X();
```

Once the recursive-descent parser has been constructed, semantic action code may be freely inserted.

5.2 Recursive-Ascent

It has been observed by several people that a directly executable style of bottom-up parser, analogous to the relationship between LL(1) and recursive-descent, can be created[3, 4, 14, 15]. The proposed schemes require the creation of one (recursive) procedure for each state in the LR parser. The actions of each procedure are determined by the LR items for the corresponding state. In general, the procedure has two parts. The first part is constructed from those items that generate *T_TABLE* actions, i.e. where the current input symbol selects a shift or a reduce

action. The second part is constructed from those items that generate *N_TABLE* actions, i.e. for *goto* transitions on non-terminal symbols. The overall structure of the procedure for a state Q is as follows.

```
void Q() {
      terminal actions;
      while (TRUE) {
            non-terminal actions
      }
}
```

If one or more items have the form $[A \rightarrow \alpha \bullet a\beta]$ then they create a shift transition in the recognizer to some state QQ on symbol a. The corresponding code in the terminal actions part of the recursive-ascent parser is as follows.

```
if (sym == a) {
      sym = next input symbol;
      QQ();
}
```

If an item has the form $[A \rightarrow \alpha \bullet\ ]$ then it corresponds to a reduce action. The action is triggered if the current symbol is a member of the context set for this reduce item. The corresponding code in the terminal actions part of the parser is as follows.

```
if (sym ∈ σ) {
      lhs = A;
      return*k;
}
```

where σ represents the context set, $k = |\alpha|$, and *lhs* is a global variable. The statement `return*`k is meant to represent a multi-level procedure return construct. The statement `return*1` causes a return to the caller of the current procedure, `return*2` returns to the caller of the caller, and so on. In addition, `return*0` is permissible and has no effect (it is equivalent to the null statement). Multi-level returns are easy to simulate if they are not provided as a construct of the programming language.

Non-terminal actions are shift actions (goto actions) and are created from items of the form $[A \rightarrow \alpha \bullet B\beta]$. If the destination state is QQ then the code is as follows.

```
if (lhs == B) QQ();
```

As a complete example, if state Q has the following set of items

$[A \rightarrow\ \bullet\, bC]$	*shift to state* Qb
$[D \rightarrow\ \bullet\, b]$	*shift to state* Qb

$[E \rightarrow BB \bullet\ ]$	*reduce, context set* = {*d*,*e*}
$[A \rightarrow B \bullet Gc]$	*shift to state* QG
$[D \rightarrow aB \bullet H]$	*shift to state* QH

(with additional information needed for the parser construction shown on the right) then the corresponding recursive ascent procedure for Q would be as follows.

```
void Q() {
     if (sym == b) {
          sym = next input symbol;
          Qb();
     } else if (sym == d || sym == e) {
          lhs = E;
          return*2;
     } else
          report a syntax error;
     while (TRUE) {
          if (lhs == G) QG();
          if (lhs == H) QH();
     }
}
```

Note: the apparently non-terminating `while` loop is eventually broken when one of the procedures QG or QH, or one of their descendant procedures, performs a multi-level return that passes control back to an ancestor of Q.

Recursive ascent parsers, however, appear to have little practical value. Three reasons can be listed. First, few people would have the inclination or the expertise to construct a recursive ascent parser by hand. Typical grammars for programming languages have recognizers with hundreds of states. Even for small grammars, manual computation of the LR(0) sets of items and their context sets is an error-prone activity. Thus, one of the main reasons for the popularity of recursive descent, the ability to code the parser manually, does not apply. Second, only an expert should attempt to insert semantic code into a recursive ascent parser manually. Not only would there be difficulty in determining correct positions for code insertions, but there is also the problem that the same semantic code may have to be replicated in many states. Third, recursive ascent parsers are not competitive with other forms of LR parsing in terms of space or speed. A recursive ascent parser does execute faster than a table-driven equivalent, but an enormous space penalty must be paid. The data in Table 1, given later, includes comparisons between a recursive-ascent parser and a table-driven LALR(1) parser for the same grammar. In any case, recursive ascent parsing should be compared against other forms of directly executable LR parsers[10, 12]. These other approaches produce parsers that are still faster than recursive ascent parsers yet they have considerably smaller space requirements.

In summary, we can think of no reason why a recursive ascent parser should be used in a

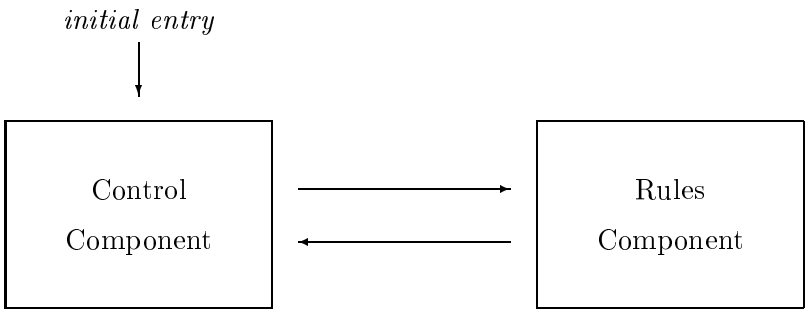

Figure 3: Two-Component Model for a Parser

compiler. However, in the next section of the paper, we will show why a directly executable version of LAXLC(1) *does* appear to be a useful and practical parsing method.

5.3 Directly-Executable Left-Corner Parsers

It is straightforward to separate the code of a recursive descent parser into two parts. One component contains a separate procedure for each production rule. If rule number 3 is, for example, $A \rightarrow aBCd$, we might construct a procedure for it like the following.

```
void Rule_3() {
     match(a);
     B();
     C();
     match(d);
}
```

where *match* is a macro or procedure that performs the task of matching the current symbol against the terminal supplied as its argument. The other component of the parser contains a separate procedure for each non-terminal symbol.

We might diagram this subdivision of the parser as in Figure 3. The point to be observed with this model of the parser is that the user can freely insert semantic code into the rules section. This is as natural as inserting code, or markers that signal semantic actions, into the original grammar and re-processing the grammar through a parser generator. Many programmers would consider it to be less trouble also.

The LAXLC(1) parser can also be implemented according to the pattern shown in Figure 3. As in the recursive descent version, the rules component of the parser can also contain one procedure for each production. If, for example, $A \rightarrow a\diamond b\diamond Cc\diamond d\diamond D\diamond$ is rule number 4 in the grammar, where

the free positions are marked, we can implement this rule by the following code.

```
void Rule_4() {
     match(b);
     Cc();
     match(d);
     D();
}
```

The *match* macro or procedure is the same as before. The procedure calls *Cc()* and *D()* invoke the control component of the parser to (recursively) begin new parses at the entry states Q_{Cc} and Q_D, respectively.

That is, the control component must implement the recognizer in a slightly different way to that described previously for a left-corner parser. Shift actions are implemented as before, causing a new state to be pushed onto the stack. However, the action for *announce rule i*, where rule i is $A \rightarrow \alpha\widehat{i}\beta$, is to pop $|\alpha|$ states from its stack exposing *top* as the top stack entry, to push $N\_TABLE[top, A]$ onto the stack, and to then call the procedure *Rule_i()*. The action for *pop k* is to pop k states from the stack and then exit from the parser, returning to the caller. (In all cases except for the very last return, this will return control back into one of the rule procedures.)

The control component of the parser may be implemented in either a table-driven manner or in the directly executable style of recursive ascent. In the latter case, we would refer to the entire parser as being recursive-ascent-descent (ascent for the control component and descent for the rules component). In practice, a table-driven implementation of the control component might be preferred because it is smaller and can perform syntax error recovery more easily.

A big advantage of the two component form of LAXLC(1) parser is that it gives the user the opportunity to insert semantic code directly into the appropriate production rules. Note that *every* free position in the grammar is available, and no non-free positions are available. Thus the user is given the freedom of recursive descent but for any LL(1) or LALR(1) grammar. The only point on which the approach suffers in contrast with recursive descent is that mechanical generation of the parser is a practical necessity. But, given a published LALR(1) grammar for a language, the unadorned grammar need be processed by the parser generator once only. From that point on, the user can incrementally insert semantic code into the rules component of the parser and need never run the parser generator again.

The recursive-ascent-descent parsing approach should also have practical value in Prolog applications. A commonly used method for syntax checking and processing input is to use a technique known as *definite clause grammars* (or DCGs)[17]. In fact, a DCG is a form of recursive descent parser. We therefore propose that the LAXLC(1) method could be used to generate parsers in Prolog. The rules component of the parser would look very similar to a DCG recognizer and the user would have the same freedom to insert semantic actions into the Prolog code.

As a final note, we should point out that a similar two-component form of XLC(1) parser can also be constructed. There is, however, a complication. The XLC(1) parser construction algorithm will usually create several versions of an entry state Q_A, one for each context in which A needs to be recognized. The control component of the parser would therefore need to perform

extra bookkeeping work to remember the current context when it calls a procedure *Rule_i* and to use this context information to select the correct version of the entry state should control recursively re-enter.

6 Experience

The LAXLC(1) construction algorithm has been used as the basis for a parser generator implementation. The parser generator normally selects the first free position in each rule as the recognition point. However the default choice can be overridden, if desired, in order to generate a pure recursive-ascent parser. The parser is normally created as two C source code files, corresponding to the control component and rules component, as explained above. By placing the rules component in a file by itself, it becomes much more amenable to manual editing. The control component can optionally be created in either a table-driven form or in the recursive ascent (pure code) style. As observed in [15], a considerable improvement in execution efficiency is possible if the patterns of control flow in the recursive ascent routines are analyzed and translated into proper loop structure. A similar technique, but based on interval analysis[1], has been incorporated into our implementation.

Using a grammar for the C language as a test case, comparative figures for five different parsers generated from the grammar are given in Table 1. Before commenting on the relative performances and sizes of the parsers, we should begin by stating how each parser was constructed. The first row of the table gives data for a recursive-ascent-descent parser where both the Control and Rules components are directly executed (indicated by the *d* entries shown for both component descriptions). The second row is similar except that the control component is implemented in a conventional table-driven manner (indicated by a *t* entry for the control component form). The third row describes a pure recursive-ascent parser, hence the control component is directly executed and the rules component is empty (indicated by '–' for its form). The fourth row is similar except that the control component is table-driven – clearly this is equivalent to a standard table-driven LALR(1) parser. For comparison purposes, a fifth row giving data for a parser generated by *yacc*[11], the standard parser-generator on the Unix system, is also provided. Differences between the fourth and fifth rows reflect only on the different representations chosen for entries in the parser tables and on different table compression techniques used. (Only a simple unsophisticated technique was used to compress the control tables whose sizes are shown in the second and fourth rows.)

7 Conclusions

One issue that is likely to be a concern is the ability to incorporate syntactic error recovery techniques into the parser. Many techniques have been developed for use with the LR methods. Recovery techniques for use with top-down methods are usually considered to be inferior because of the difficulty of making contextual information available to the parser. What is possible with the left-corner parsers as described here?

Parser	Parser Form		Size (Source Lines)		Size (Object Bytes)		Speed
	Control	Rules	Control	Rules	Control	Rules	
1.	d	d	6273	1193	25032	8496	2.71
2.	t	d	697	1193	8938	8496	1.03
3.	d	–	12639	0	59704	0	2.81
4.	t	–	1055	0	14240	0	1.26
YACC	t	–	601	0	5832	0	1.00

Table 1. Comparisons between Parsers

If a table-driven implementation of the control part of the left-corner parser is used, it is possible to simulate an equivalent of the state stack that would be maintained by a LR parser. It is possible even if the rules component of the left-corner parser is directly executed. Thus, when a syntax error is detected, exactly the same information can be made available to the recovery algorithm as would be available to any LR recovery method. Implementing the recovery actions that the algorithm requires is more problematic, however. If the rules component of the parser is directly executed, some kinds of recovery action may be difficult or impossible to perform. A reasonable suggestion, therefore, is to restrict our attention to strategies whose recovery actions are to skip input symbols and/or to insert new symbols in the input stream. One such technique was devised for LR parsers by Röhrich[16] and could easily be adapted for use with our left-corner parsers. An alternative possibility is to augment the grammar with error productions and use a technique similar to that employed in yacc[11].

We conclude by summarizing the advantages of the left-corner parsing methods described here. They provide the best of both worlds in combining the full power of the bottom-up LR methods with the ease of use of the top-down recursive-descent method. Every position in a rule where a semantic action may be inserted is available to a compiler developer without re-running a parser generator. The size of the parser and its speed are similar to those observed with other parsing methods. Finally, a multiple-entry parser is automatically generated, with an entry point (or entry parameter) for every non-terminal symbol in the grammar that can be handled meaningfully.

Appendix

The result that the LAXLC(1) construction algorithm creates correct, deterministic, recognizers for LALR(1) grammars is based on the following sequence of steps. First, every production in a LALR(1) grammar has a free position at its right-hand end (Lemma 1). Second, the LAXLC(1) construction algorithm is identical to the LALR(1) construction algorithm if all rule recognition points are set at the ends of the right-hand sides (Lemma 2). Third, if the recognition point of one rule is moved left to the next free position (assuming there is one), the recognizer constructed by the LAXLC(1) algorithm is deterministic and accepts the same language as before. Fourth, a series of movements of the recognition points to the left can achieve any allowable configuration

of recognition point positions.

The first two steps of the argument correspond to the following two lemmas.

Lemma 1 Every production in a LALR(1) grammar has a free position at its right-hand end.

Proof It is straightforward to show that the Purdom-Brown algorithm for finding free positions [13, Algorithm 2] must mark the right-hand position of every rule as free if the recognizer is deterministic. This is true by definition for a LALR(1) recognizer generated from a LALR(1) grammar. Finally, [13, Theorem 13] proves the correctness of the marking algorithm.

Lemma 2 The LAXLC(1) construction algorithm generates the same recognizer as the LALR(1) construction algorithm if the recognition point of each rule is selected to be at the end of the rule's right-hand side.

Proof We will first compare sets of items generated by the two construction algorithms, ignoring context set differences. For the purposes of comparing an item used by the LAXLC(1) algorithm with an item used by the LALR(1) algorithm, the recognition point marker should be ignored.

We observe that the dot marker in an item cannot coincide with the rule's recognition point if the dot precedes a symbol (either a terminal or a non-terminal). Thus, the LAXLC(1) closure function, *cl*, will produce the same result as the standard LR(0) closure operation. Similarly, the LAXLC(1) goto function will produce the same set of items for the destination state.

The start states of the two recognizers differ in a very minor way. For a grammar G = $< V_N, V_T, P, S >$, the LAXLC(1) recognizer has a start state, Q, formed by $cl(\{\,[\,S_S \rightarrow \vdash \bullet\ S\,]\,\})$ whereas the LALR(1) algorithm given by [1] creates its start state, Q′, from $cl(\{[\,S' \rightarrow \bullet\ S\,]\}\,)$, where $S' \rightarrow S$ is an additional rule (i.e., an *augmented grammar* is used). If we make the obvious correspondence that $[\,S_S \rightarrow \vdash \bullet\ S\,] \equiv [\,S' \rightarrow \bullet\ S\,]$, and $[\,S_S \rightarrow \vdash S \bullet\ ] \equiv [\,S' \rightarrow S \bullet\ ]$, then the two recognizers have identical sets of items and have identical transitions between states.

Next, we can consider the context sets associated with items in the two recognizers. We may ignore the rule that applies to context sets for additional entry states in the LAXLC(1) recognizer, since there are no additional entry states created. In this case, the given LAXLC(1) context set inclusion rules correspond to the algorithm in [1, Fig. 4.43] for computing LALR(1) context sets.

Finally, we observe two points. When the recognition point of rule i is at the end of its right-hand side, the effect of the action *announce rule i* is identical to that of a LALR(1) recognizer performing the action *reduce by rule i*. Second, the LAXLC(1) recognizer will have a *pop* action in only one state, namely in the state formed by goto(Q,S). This will be a *pop 2* action and, if executed, it must cause the stack to become empty and hence cause the recognizer to halt. The LALR(1) recognizer will have the action *reduce by rule* $S' \rightarrow S$ instead, and its effect is also to cause the recognizer to halt.

Therefore we conclude that the LAXLC(1) and LALR(1) recognizers perform identical parsing actions on all input sentences. They are equivalent.

The third step of the argument is proved by Theorem 1, below. This theorem is based on a comparison of states and their sets of items for two LAXLC(1) recognizers. The first recognizer,

M, is a deterministic LAXLC(1) recognizer constructed from grammar G $= < V_N, V_T, P, S >$. In G, we assume that rule i is $A \rightarrow \alpha \diamond X_1, X_2, ...X_k \, \hat{\imath} \, \gamma$, where $k > 0$, $\diamond$ marks a free position, each position between X_i and X_{i+1} is not free for $1 \leq i < k$, and $\hat{\imath}$ marks the recognition point of rule i. The second recognizer, M$'$, is constructed from grammar G$'$. G$'$ is identical to G except that rule i is replaced by $A \rightarrow \alpha \, \hat{\imath} \, X_1, X_2, ...X_k \, \gamma$.

When comparing an item in a state of M with an item used in constructing M$'$, we ignore the presence of recognition point markers in items. For example, the items $[\, A \rightarrow \alpha \bullet B \, \hat{\imath} \, \gamma \,]$ and $[\, A \rightarrow \alpha \bullet \hat{\imath} B \, \gamma \,]$ would be taken to be equivalent.

We need to define one term that is used in the main theorem.

Definition Consider a start state Q_S and a cycle-free path that begins at state Q_S and ends at a state Q. Each edge in the path corresponds to a transition on some terminal or non-terminal symbol. The sequence of symbols along the path will be referred to as an *accessing sequence* for state Q relative to start state Q_S.

Theorem 1 M$'$ is a deterministic recognizer that accepts the same language as M.

Outline of Proof We begin by comparing the states of M and M$'$. In the following, any state name that has a prime, such as Q$'$, refers to a state in M$'$; similarly, a state whose name lacks a prime belongs to M. We consider a state in M and a state in M$'$ to be equivalent if they have equivalent sets of items. (The context sets associated with the items are ignored.)

Since G is reduced, M must contain at least one state that includes the item

$$[\, A \rightarrow \alpha \bullet X_1 X_2 ... X_k \, \hat{\imath} \, \gamma \,].$$

Call one such state Q and let the accessing sequence of Q be κ. M must also contain an additional k states Q_1, Q_2 ... Q_k, reached from Q by following a transition labelled X_1, then a transition labelled X_2, and so on. State Q_j, $1 \leq j \leq k$, includes $[\, A \rightarrow \alpha X_1 ... X_j \bullet X_{j+1} ... X_k \, \hat{\imath} \, \gamma \,]$ as a core item.

M$'$ must contain a state, Q$'$, that has the same accessing sequence, κ, as Q and Q$'$ necessarily includes the item $[\, A \rightarrow \alpha \bullet \hat{\imath} \, X_1 X_2 ... X_k \, \gamma \,]$. The LAXLC(1) rule for construction of entry states forces M$'$ to have an entry state with the single basis item $[\, S_\beta \rightarrow \vdash \bullet X_1 X_2 ... X_k \,]$. Call this entry state Q'_0. M$'$ must have an additional k states Q'_1, Q'_2 ... Q'_k reached from Q'_0 by following a transition labelled X_1 from Q'_0, then a transition labelled X_2, and so on. State Q'_j, $1 \leq j \leq k$, has $[\, S_{X_1 ... X_k} \rightarrow \vdash X_1 X_2 ... X_j \bullet X_{j+1} ... X_k \,]$ as a core item. These states in M and M$'$ are diagrammed in Figure 4.

The assumption that a free position is located between α and X_1 in the rule $A \rightarrow \alpha X_1 ... X_k \gamma$ implies that state Q_1 has only a single core item, the one shown in Figure 4. The LAXLC(1) construction rules cause the corresponding state in M$'$, Q'_1, to also have the single core item shown. If we take $[\, A \rightarrow \alpha X_1 ... X_j \bullet X_{j+1} ... X_k \, \hat{\imath} \, \gamma \,]$ and $[\, S_{X_1 ... X_k} \rightarrow \vdash X_1 ... X_j \bullet X_{j+1} ... X_k \,]$ as being equivalent items, then it follows that (1) the states Q_j and Q'_j, $1 \leq j \leq k$ are equivalent, and (2) Q is equivalent to $Q' \cup Q'_0$. Furthermore, transitions labelled by a particular symbol from equivalent states in the two machines lead to equivalent states.

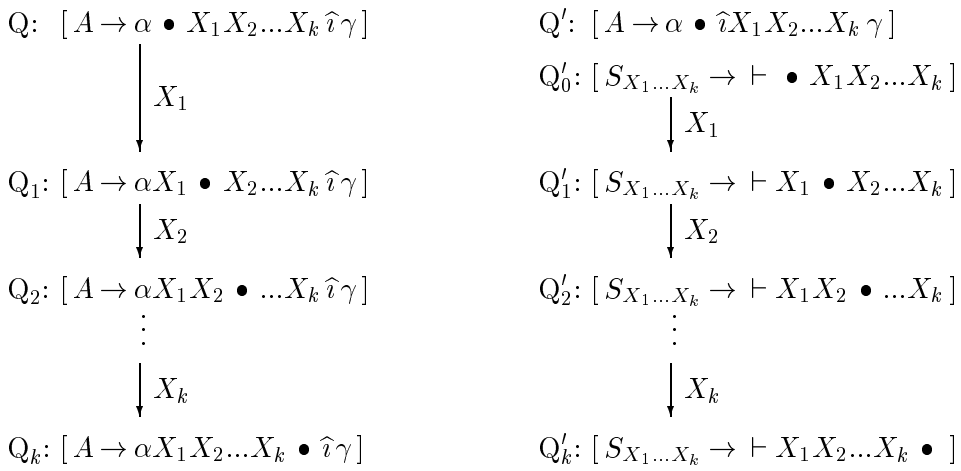

Figure 4: Equivalent States in M and M′

Next, we can show that the two context sets

$$context(\mathrm{Q}, [A \to \alpha \bullet X_1 ... X_k \, \hat{\imath} \, \gamma])$$

and

$$context(\mathrm{Q}', [A \to \alpha \bullet \hat{\imath} \, X_1 ... X_k \gamma])$$

must be equal. Let σ denote these context sets. Then we must have

$$\mathrm{first}_1(\gamma, \sigma) \quad \subseteq \quad context(\mathrm{Q}'_0, [S_{X_1 ... X_k} \to \vdash \bullet X_1 ... X_j X_{j+1} ... X_k])$$

Equality of the sets does not necessarily hold because there may be states other than Q in M that contain the item $[A \to \alpha \bullet X_1 ... X_k \, \hat{\imath} \, \gamma]$ and because state Q'_0 in M′ may be required for some rule other than rule i.

The rules for propagation of context sets imply that reduce items for states in M that are reachable from Q must be subsets of the context sets for the equivalent reduce items in M′. The structural equivalence of the two machines plus the inclusion property for the context sets proves the theorem in one direction. Namely, every sentence that can be recognized by M must also be recognizable (perhaps non-deterministically) by M′.

The converse problem, that of showing that every sentence recognizable by M′ is also recognized by M, must now be considered. The structural equivalence of the two machines implies that any sentence recognized by M′ but not by M must be involve non-deterministic recognition. That is, one or more states in M′ must contain conflicts. Such a conflict must occur in Q'_0 or in

one of its successor states. Let x denote one of the conflicting lookahead symbols. If the conflict is a reduce-reduce, x occurs in the context sets of both reduce items. Otherwise, if the conflict is a shift-reduce, x occurs in the context set of the reduce item. The x symbol in the context sets must have been propagated according to the rules in section 4.2. The symbol cannot have been *spontaneously* generated (to use the terminology of [1]) in any of the states Q'_i, $0 \leq i \leq k$, otherwise the same conflict would have to exist in the equivalent state of M. Thus one of these states must have inherited the symbol from an external predecessor. If this external predecessor is *not* a state containing the item $[A \rightarrow \alpha \bullet \widehat{\imath} X_1 ... X_k \gamma]$ (without loss of generality, assume this state is Q') then, again, the same conflict would have to exist in M. The only possibility is that x is inherited from Q' according to the LAXLC(1) rule for context set propagation. But Q and Q' have the same context sets and therefore propagation from Q to Q_1, and so on, should lead to the equivalent conflict on x in M.

Corollary The LAXLC(1) construction algorithm generates a deterministic recognizer for any LALR(1) grammar.

References

[1] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA (1985).

[2] Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translation and Compiling*, Vol. I: *Parsing.* Prentice-Hall, Englewood Cliffs, NJ (1972).

[3] Aretz, F.E.J.K. On a Recursive Ascent Parser. Information Processing Letters, **29,3** (Nov. 1988), pp. 201-206.

[4] Barnard, D.T., and Cordy, J.R. SL Parses the LR Languages. Computer Languages **13,2** (1988), pp. 65-74.

[5] Beatty, J.C. On the Relationship Between LL(1) and LR(1) Grammars. Journal of ACM, **29,4** (Oct. 1982), pp. 1007-1022.

[6] Chapman, N.P. *LR Parsing: Theory and Practice.* Cambridge University Press, Cambridge, U.K. (1987).

[7] Demers, A.J. Generalized Left Corner Parsing. Proc. 4th Symposium on Principles of Programming Languages (1977), pp. 170-182.

[8] DeRemer, F.L. Simple LR(k) Grammars. Communications of ACM, **14,7** (July 1971), pp. 453-460.

[9] Fischer, C.N., and LeBlanc Jr., R.J. *Crafting A Compiler.* Benjamin/Cummings, Menlo Park, CA (1988).

[10] Horspool, R.N., and Whitney, M. Even Faster LR Parsing. Software – Practice & Experience **20,6** (June 1990), pp. 515-535.

[11] S.C. Johnson, 'YACC – Yet Another Compiler Compiler', *UNIX Programmer's Manual, 7th Edition*, **2B**, (1979).

[12] Pennello, T.J. Very Fast LR Parsing. Proc. 1986 Symposium on Compiler Construction, ACM SIGPLAN Notices **21,7** (1986), pp. 145-151.

[13] Purdom, P., and Brown, C.A. Semantic Routines and LR(k) Parsers. Acta Informatica, **14** (1980), pp. 299-315.

[14] Roberts, G.H. Recursive Ascent: An LR Analog to Recursive Descent. ACM SIGPLAN Notices, **23,8** (Aug. 1988), pp. 23-29.

[15] Roberts, G.H. Another Note on Recursive Ascent. Information Processing Letters, **32,5** (Sept. 1990), pp. 263-266.

[16] Röhrich, J. Methods for the Automatic Construction of Error Correcting Parsers. Acta Informatica, **13,2** (1980), pp. 115-139.

[17] Sterling, L., and Shapiro, E. *The Art of Prolog.* MIT Press, Cambridge, MA (1986).