Chapter 3

LALR(k) Lookahead Sets with Varying-Length Strings

Perhaps the most important reason for using LALR(k) grammars is that they allow the user to express certain syntactic constructs in a more intuitive manner. To motivate the importance of this observation, consider the grammar of Figure 3.1(a). This very simple definition, which clearly captures the syntactic rules of BNF, is LALR(2). To see this, observe that when parsing the right-hand side of a rule, one cannot determine when looking at a symbol whether or not it belongs to the current rule, or if it is the left-hand side symbol of the next rule.

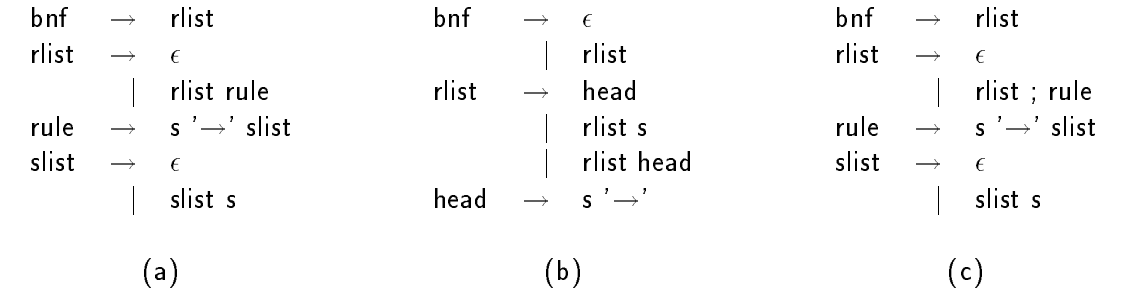

Figure 3.1: LALR grammars for BNF

The grammar of Figure 3.1(b) is an LALR(1) grammar for the same language, but note that the rules of that grammar do not accurately reflect the syntactic structure of a BNF rule. As a result when using a parser generated from that grammar, there is no convenient way of determining when the end of a rule (from the input) has been reached.

The use of multiple lookahead symbols is also helpful in detecting certain common errors that can be flagged as warnings. For example, most programming languages contain "dead" keywords, such as THEN, and separators, such as ";" whose only purpose is to separate syntactic constructs. It is usually the case that if the parser can look ahead at more than one symbol, then it can determine without the special markers where a particular

construct ends and another begins. In such a case, the markers can be made optional (in the grammar definition) and a semantic "warning" message issued when the "empty" choice is reduced. For example, each occurrence of "THEN" in a typical programming language grammar can be replaced by a nonterminal "then" which is defined as follows:

$$\begin{array}{lcl} \mathsf{then} & \rightarrow & \epsilon \\ & | & \mathsf{THEN} \end{array}$$

Similarly, observe that the grammar of Figure 3.1(a) can be modified into the LALR(1) grammar of Figure 3.1(c) by introducing ";" as a marker symbol to separate adjacent rules. When processing the right-hand side of a rule with that grammar, the parser always shifts on "s" and reduces by "rule" upon encountering a ";" or the end-of-file token.

In the area of parser generation, the main innovation of this thesis is a new algorithm for the construction of efficient LALR(k) parsers for values of k larger than 1. As noted earlier, for practical reasons, such a parser cannot be efficiently constructed in the usual fashion (i.e., first construct an LR(0) automaton and then resolve each conflict by computing the relevant LALR(k) lookahead sets). Instead, the minimum amount of lookahead information is computed in an incremental fashion, when needed, as follows. When conflicts are detected in an LR(0) state, the parser generator first computes the relevant LALR(1) lookahead set for each final item in that state. If these lookahead sets are sufficient to resolve the conflicts, the generator goes no further. Otherwise, each conflict symbol is extended into a set of lookahead strings of length 2. If that does not resolve all the remaining conflicts, each conflict string of length 2 is extended into a set of lookahead strings of length 3, and so on, until either the upper limit k is reached or all conflicts are successfully resolved. At the end of this process, each final item in an inconsistent LR(0) state is associated with a lookahead set of strings whose length vary from 1 to k.

The use of lookahead sets with variable-length strings raises an important question, namely, what is a good representation for the parsing tables in such a framework? Clearly, an ACTION matrix whose columns consist of all terminal strings of length 1 up to k would be even more space-consuming than the standard representation. A suitable representation will be described later, but first, the traditional parsing table representation and the concept of a *default action* are reviewed.

Consider the grammars (a) and (b) of Figure 3.1. Figure 3.2 shows possible parsing tables for (a) and Figure 3.3 shows possible parsing tables for (b). Observe that for an LALR(1) parser (Figure 3.3), each column of its ACTION matrix is indexable by a single terminal symbol. Thus, the terminal transitions of the GOTO matrix can be combined with the ACTION matrix to obtain a merged table as in Figure 3.4. Furthermore, note that some states contain many reduce actions by the same rule. These actions can be factored into a single *default reduce* action by adding a *default* column (?) to the ACTION matrix as shown in Figure 3.5. In general, for a given state of an LALR(k) parser, the rule with which the most reduce actions are associated is chosen as the default action [33].

During parsing, if the ACTION matrix entry for a given (state, symbol) pair is **error**, the default value associated with the pair (state, ?) is used. When the default action for a given state is not **error**, it may allow the parser to incorrectly perform a reduce action. However, in such a case, the error will be detected later since the parser will not be able to shift on the input symbol in question.

	ACTION									GOTO					
	→→	→s	→⊥	s→	ss	s⊥	⊥→	⊥s	⊥⊥	→	s	bnf	rlist	rule	slist
1				R2					R2			3	2		
2				S					R1		4			5	
3									acc						
4		S	S							6					
5				R3					R3						
6				R5	R5	R5			R5						7
7				R4	S	S			R4		8				
8				R6	R6	R6			R6						

Figure 3.2: LALR(2) parsing tables for BNF grammar (a)

	ACTION			GOTO					
	→	s	⊥	→	s	bnf	rlist	rule	slist
1		S	R1		3	4	2		5
2		R2	R2						
3	S			6					
4		S	acc		8				7
5		R3	R3						
6		R6	R6						
7		R5	R5						
8	S		R4	6					

Figure 3.3: LALR(1) parsing tables for BNF grammar (b)

	ACTION			GOTO			
	→	s	⊥	bnf	rlist	rule	slist
1		S3	R1	4	2		5
2		R2	R2				
3	S6						
4		S8	acc				7
5		R3	R3				
6		R6	R6				
7		R5	R5				
8	S6		R4				

Figure 3.4: Merged LALR(1) parsing tables for grammar (b)

	ACTION				GOTO			
	?	→	s	⊥	bnf	rlist	rule	slist
1	R1		S3		4	2		5
2	R2							
3		S6						
4			S8	acc				7
5	R3							
6	R6							
7	R5							
8	R4	S6						

Figure 3.5: Merged LALR(1) parsing tables with default actions for grammar (b)

Once again, consider the grammar of Figure 3.1(a) and its parsing tables (Figure 3.2). With such an ACTION matrix, the parser always needs access to the next two symbols in the input in order to make a parsing decision. However, notice that the only time the two lookahead symbols are necessary is when the parser is in state 7 and the next input symbol is s. In that case, if the successor of s is another s or $\perp$, the parser shifts; if the successor is a $\rightarrow$, the parser reduces. (If default actions are used in this table, the parser would always reduce if the successor is neither s nor $\perp$.)

Just as the lookahead set for a LALR(k) parser can be constructed in an incremental fashion, the parser itself can "lookahead" in an incremental fashion given a proper representation for the ACTION matrix. That is, at run time, the parser can behave just like a LALR(1) parser when only a single lookahead is sufficient and use extra lookahead when required. This is achieved by dividing the traditional ACTION matrix into two separate tables: an ACTION_1 matrix indexable by (state, terminal) pairs, just like an LALR(1) ACTION matrix, and a *lookahead table*. Each entry of the ACTION_1 matrix contains either a relevant action to be performed (**shift** q**, reduce** i**, accept, error**) or it indicates that more lookahead is required (**lookahead** q') and identifies where in the lookahead table to begin the search.

Additional lookahead is required for an LALR(k) parser when for a given state q and terminal symbol a, the ACTION matrix of the parser contains two or more different useful entries on lookahead strings that start with a. Let the set of terminals represent an alphabet. A deterministic finite automaton (DFA) can be constructed in a straightforward manner to recognize a given set of strings on that alphabet. For example, consider a set of strings $L = \{abc, adb, adc, acb\}$ on the English alphabet. Figure 3.6 shows a DFA to recognize this set of strings.

Assume that the set $\{a, b, c, d\}$ is a subset of the set of terminals of a grammar G and that the set L above is a set of conflicting lookahead strings in a state q of an LALR(k) parser for G. A DFA similar to the one in Figure 3.6 (called a *lookahead DFA*) can be constructed to recognize the elements of L. The initial state q_1 of the lookahead DFA is the inconsistent state q of the parser. The other states of the DFA are new states called

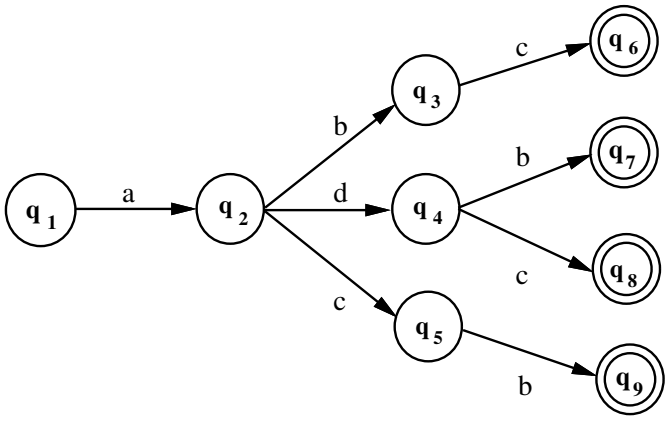

Figure 3.6: DFA for {abc, adb, adc, acb}

---

lookahead states (different from the lookahead states of [6]). Each path in the DFA from the initial state q_1 to some final state q_f spells a relevant lookahead string x in the conflicting set and q_f is associated with the LALR(k) action of the pair (q, x). A transition into a lookahead state is called a *lookahead shift*.

A lookahead DFA can be stored in a table by associating each of its non-final states with a vector (row) indexable by terminal symbols. Each entry in such a vector contains either a transition into another lookahead state q' or an actual parsing action. A transition into a lookahead state q' is denoted: **lookahead shift** q'. The lookahead table of an LALR(k) parser is a matrix formed with the rows associated with lookahead states.

At run time, when the parser enters state q (in the example above), if the next input symbol is a, the ACTION$_1$ matrix yields a lookahead shift action to the successor of q on a. A lookahead shift is different from a normal shift (into an LR(0) state) in that it instructs the parser to look at the next symbol in the input without consuming the current symbol. Once the DFA is entered, the parser tries to match the next symbols in the input with a valid path in the DFA. If successful, a final state q_f is reached and the action associated with q_f is performed; otherwise, the **error** action is computed at the first illegal combination of (state, symbol) pair encountered.

Assume that each string element of the set L is associated with a parsing action in state q as follows: ACTION(q,abc)=S2; ACTION(q,abd)=R5; ACTION(q,adc)=R6; ACTION(q,acb)=R5. The lookahead DFA for ACTION$_1$(q,a) is shown in Figure 3.7. Observe that once states q_3 and q_5 are entered, the DFA can only follow a single path to a final state. When that is the case, if the action associated with the final state is a reduce action, all states on that path can be removed (except for the final action). This is analoguous to the default reduce optimization mentioned earlier. Figure 3.8 shows a DFA for the actions above with default reductions. When default actions are used in a lookahead DFA, some of the paths from the starting state to a final action (and the lookahead strings that spell these paths) may be shorter than k; hence, the term: *lookahead sets with variable-length strings*.

Recalling the earlier informal discussion of how conflicts are resolved in this method,

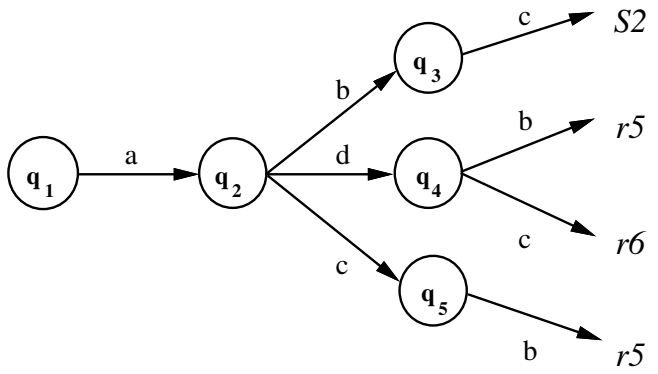

Figure 3.7: Lookahead DFA

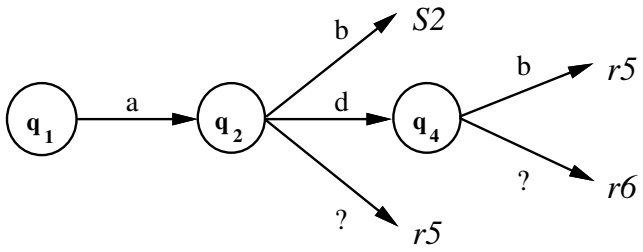

Figure 3.8: Lookahead DFA with default

		ACTION				**GOTO**			
		?	→	s	⊥	bnf	rlist	rule	slist
	1	R2				3	2		
	2	R1		S4				5	
	3				acc				
$ACTION_1$:	4		S6						
	5	R3							
	6	R5							7
	7	R4		L9					
	8	R6							
Lookahead:	9	R4		S8	S8				

Figure 3.9: LALR(2) parsing tables with variable lookahead strings for grammar (a)

for each inconsistent state q, it is precisely the DFAs with default actions that are constructed incrementally for each symbol a that is in conflict in q, and "grafted" onto q. Thus, the space required to construct a LALR(k) parser with variable lookahead strings is proportional to the size of the resulting parser. Consequently, this method is both fast and practical. Figure 3.9 shows the parsing tables with variable lookahead strings for the LALR(2) grammar of Figure 3.1(a).

3.1 LALR(k) Lookahead Sets

In this section, the fundamental definitions of READ and FOLLOW sets are extended to accomodate the general case of $k \geq 0$.

Definition 3.1.1

$\text{FOLLOW}_0(p, A) = \{\varepsilon\}$
$\text{FOLLOW}_k(p, A) = \text{LALR}_k(p, [A \rightarrow \cdot\omega])$

Lemma 3.1.1

$\text{FOLLOW}_k(p, A) =$
$\bigcup\{\text{FIRST}_k(\beta) \oplus_k \text{FOLLOW}_k(p', B) \mid [B \rightarrow \alpha \cdot A\beta] \in p, \text{and } p' \in \text{PRED}(p, \alpha)\}$

Lemma 3.1.2

$$\text{LALR}_k(q, A \rightarrow \omega\cdot) = \bigcup\{\text{FOLLOW}_k(p, A) \mid p \in \text{PRED}(q, \omega)\}$$

Lemmas 3.1.1 and 3.1.2 follow directly from the two Lemmas 2.2.2 and 2.2.1, respectively, of Kristensen and Madsen stated earlier.

Definition 3.1.2

$$\text{READ}_k(p, X) = \bigcup\{w \mid w \in \text{FIRST}_k(\beta),\ |w| = k,\ [B \rightarrow \alpha \cdot X\beta] \in p\}$$

Definition 3.1.3

$$\text{SHORT}_k(p, X) = \bigcup\{[w, [B \rightarrow \alpha \cdot X\beta]] \mid w \in \text{FIRST}_k(\beta),\ 0 < |w| < k,\ [B \rightarrow \alpha \cdot X\beta] \in p\}$$

Definition 3.1.4

$$\text{READ}_{k*}(p, X) = \text{READ}_k(p, X) \cup \{w \mid [w, [B \rightarrow \alpha \cdot X\beta]] \in \text{SHORT}_k(p, X)\}$$

For a given state p and symbol X, $\mathrm{READ}_k(p, X)$ is the set of all strings of length k that can be read following a transition on X in p; $\mathrm{SHORT}_k(p, X)$ is the set of pairs $[w, [B \to \alpha \cdot X\beta]]$ where $[B \to \alpha \cdot X\beta]$ is an item in the state p and w is a non-empty string of length less than k that is derivable from β; $\mathrm{READ}_{k*}(p, X)$ is simply the union of the sets $\mathrm{FIRST}_k(\beta)$ for all items $[B \to \alpha \cdot X\beta]$ in p less the empty string: ε. The set of strings in $\mathrm{READ}_{k*}(p, X)$ will be referred to as strings of length k^*. Combining Lemma 3.1.1 and Definitions 2.2.3, 3.1.2 and 3.1.3, one obtains the following lemma:

Lemma 3.1.3
$\mathrm{FOLLOW}_k(p, A) = \mathrm{READ}_k(p, A)$
$\quad \cup \bigcup\{\mathrm{FOLLOW}_k(p', B) \mid (p, A)$ **includes** $(p', B)\}$
$\quad \cup \bigcup\{\{w\}.\mathrm{FOLLOW}_{k-|w|}(p', B) \mid [w, [B \to \alpha \cdot A\beta]] \in \mathrm{SHORT}_k(p, A),\ p' \in \mathrm{PRED}(p, \alpha)\}$.

Lemmas 3.1.3 breaks down the FOLLOW_k sets into three disjoint sets. The first set is the set of strings of length k that can be read immediately after a transition in p on A. The second set consists of strings of length k that are contributed by other FOLLOW_k sets when A can be followed by a nullable suffix in p. Finally, the third set consists of strings formed by the concatenation of strings of length less than k that are derived from a suffix following A in p with all possible suffixes of the right length that may follow these short strings in the given context. Lemmas 3.1.3 and Lemma 3.1.2 show that (just as in the case of $k = 1$) the computation of LALR_k lookahead sets can be broken down into the computation of smaller components; i.e., LALR_k sets are computed using FOLLOW_k sets which, in turn, are computed using READ_{k*} sets.

3.2 Computation of READ_{k^*}, FOLLOW_k and LALR_k Sets

Let *stack* be a non-empty sequence of states representing a path in the LR(0) automaton of a grammar G and let X be a grammar symbol in G, $X \neq \epsilon$ and $X \neq S'$. A string w is said to be readable in the context of $(stack, X)$, if Xw can be successfully parsed starting from the state p on top of *stack* and no reduction on some symbol in w ever causes a *stack underflow* or consults the first element of *stack*. For the remainder of this section, a pair $(stack, \alpha)$, where *stack* is a sequence of states, will be loosely referred to as a *configuration*. When the string α in a configuration consists of a single terminal, the configuration is called a *terminal configuration*.

Since the LR(0) automaton of a context-free grammar G, LRM_0^G, is a correct parser for G (although it may be nondeterministic), any string w that is readable in the context of a configuration $([p_1, p_2, \ldots, p_n], X)$ can be parsed by executing the correct sequence of moves of the automaton starting with the configuration $([p_1, p_2, \ldots, p_n, q], w)$, where q is the state that is entered after a transition on X in p_n. In state q, there are three possibilities to consider:

1. State q contains transitions on terminal symbols. Each terminal symbol a that is directly readable in q is a possible starting symbol for one or more strings readable in the context of $(stack, X)$. The set of all suffixes of length $(k-1)^*$ that can follow a is computed, recursively, by considering the pair consisting of the state sequence $stack + [q]$ and the symbol a.

2. State q contains transitions on nullable nonterminals. After entering state q, the parser can make a transition on a nullable nonterminal without consuming any input symbol. All strings of length k^* that are readable after such a transition must also be included in the final result.

3. State p contains one or more items of the form $[C \rightarrow \gamma \cdot X]$. For each such item, after making the transition on X into q, the parser can immediately reduce by the rule $C \rightarrow \gamma X$. The reduce action consists of popping the elements of the *stack* corresponding to γX, making a transition on C and continuing the parse. Once again, after executing the reduction, the parser still has not consumed any symbol. Thus, every string of length k^* that is readable in the new context must be considered. The new context consists of the prefix of the stack obtained after the popping of γX and the symbol C.

From now on, let ts denote the state on top of *stack*; i.e., $ts = stack(\#stack)$. The following equations formally capture the set of strings of length k^* that can be read in the context of $(stack, X)$:

$$
\begin{aligned}
&\mathrm{READ}_{0^*}(stack, X) = \{\varepsilon\} \\
&\mathrm{READ}_{k^*}(stack, X) = \\
&\qquad \bigcup\{\{a\}.\mathrm{READ}_{(k-1)^*}(stack + [q], a) \mid a \in \mathrm{DR}(ts, X),\ q = \mathrm{GOTO}_0(ts, X)\} \\
&\quad \cup \bigcup\{\mathrm{READ}_{k^*}(stack + [q], Y) \mid (ts, X)\ \textbf{reads}\ (q, Y)\}. \\
&\quad \cup \bigcup\{\mathrm{READ}_{k^*}(stack(1..\#stack - |\gamma|), C) \mid [C \rightarrow \gamma \cdot X] \in ts,\ |\gamma| + 1 < \#stack\}.
\end{aligned}
$$

The solution set of interest is the smallest set that satisfies the above equations. The set $\mathrm{READ}_{k^*}(p, X)$ is simply the set of strings of length k^* that can be read in the context of the configuration $([p], X)$. The equations above can be modified to compute $\mathrm{SHORT}_k(p, X)$ but as will be shown later this set can be computed as a side-effect in an incremental algorithm for computing $\mathrm{READ}_{k^*}(p, X)$.

In practice, it is not always possible to simulate all possible steps of an LR(0) parser, as suggested above, if the underlying grammar is ambiguous. More precisely, if the LR(0) automaton contains one or more loops labeled with nullable nonterminals or the grammar contains rules that can cause a derivation of the form $A \Rightarrow^+_{rm} A$, the simulation can get into an infinite loop. These two problems are illustrated in the subgraphs of Figure 3.10. In both cases, assume that $C\gamma \Rightarrow^+ \varepsilon$. In Figure 3.10(a), once state p is entered, the parser can, without consuming any input symbol, enter state q and traverse the loop $q..r$ any number of times before moving on to state s. Similarly, in Figure 3.10(b), if the parser enters state p where it recognizes a B without consuming any input symbol, it can: enter state q, produce $C\gamma$, and reduce $BC\gamma$ to B which would bring it back to state p where the same process can be repeated indefinitely.

An algorithm can be devised to keep track of all configurations already seen, while simulating the parser, in order to avoid these two problems [24]. However, from a practical point of view, since the main goal of computing READ_{k^*} is to construct an LALR(k) parser, and since any occurrence of one of these two conditions renders a grammar *not-LR(k) for any k*, one can simply ensure that these two conditions do not occur in the given grammar and its automaton before attempting to compute the READ_{k^*} sets. From

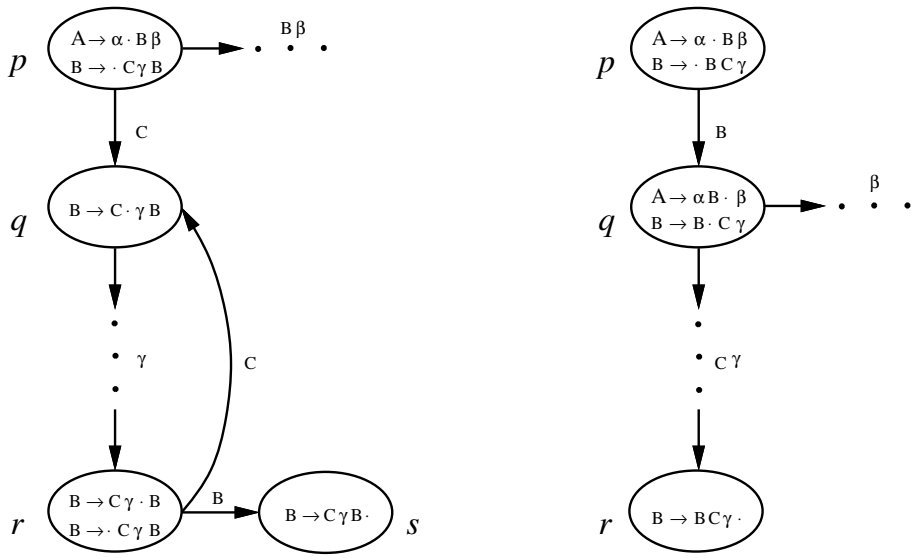

Figure 3.10: Subgraph of LR(0) automata with parsing cycles

```
function READ*(stack, X, k);
    if k = 0 then
        return {ε};
    end if;
    rd := ∅;
    ts := stack(#stack);
    q := GOTO0(ts, X);
    for a ∈ DR(ts, X) loop
        rd := rd ∪ {a.x | x ∈ READ*(stack+[q], a, k-1)};
    end loop;
    for (q, Y) | (ts, X) reads (q, Y) loop
        rd := rd ∪ READ*(stack + [q], Y, k);
    end loop;
    for [C → γ· X] ∈ ts | C ≠ S' and |γ|+1 < #stack loop
        rd := rd ∪ READ*(stack(1..#stack - |γ|), C, k)}
    end loop;
    return rd;
end READ*;
```

Figure 3.11: Recursive $READ_{k*}$ function

```
function READ_STEP(stack, X);
    configs := ∅;
    ts := stack(#stack);
    q := GOTO_0(ts, X);
    for a ∈ DR(ts, X) loop
        configs := configs ∪ {[stack+[q], a]};
    end loop;
    for (q, Y) | (ts, X) reads (q, Y) loop
        configs := configs ∪ READ_STEP(stack+[q], Y);
    end loop;
    for [C → γ·X] ∈ ts | C ≠ S' and |γ| + 1 < #stack loop
        configs := configs ∪ READ_STEP(stack(1..#stack - |γ|), C);
    end loop;
    return configs;
end READ_STEP;

function READ_*(stack, X, k);
    if k = 0 then
        return {ε};
    end if;
    rd := ∅;
    for [stk, a] ∈ READ_STEP(stack, X) loop
        rd := rd ∪ {a.x | x ∈ READ_*(stk, a, k-1)}
    end loop;
    return rd;
end READ_*;
```

Figure 3.12: Incremental READ_{k*} function

now on, it will be assumed that these conditions have been checked and that the grammar in question contains no nonterminal that can rightmost produce itself and the LR(0) automaton contains no nullable cycles. The algorithm of Figure 3.11 is a straightforward implementation of the READ_{k*} equations.

The algorithm of Figure 3.12 is an incremental version of the algorithm of Figure 3.11. Each incremental "step" is performed by a function READ_STEP which given a configuration $(stack, X)$ yields a set of new terminal configurations that can be reached after the parser has executed the transition on X in the context of $stack$. The READ_{k*} set for a given configuration $(stack, X)$ can be computed by making successive calls to READ_STEP to extend READ_{k*} strings until they reach the proper length or cannot be extended. This incremental approach has the added advantage that it can be used to simultaneously compute the set $\text{SHORT}_k(p, X)$. When invoked with a pair $([p], X)$, if the incremental READ_* function of Figure 3.12 reaches a configuration $(stack, Y)$ that is blocked because its next action is a reduction that would cause it to use $stack(1)$ or cause a stack underflow, this indicates that state p contains one or more items of the form $[C \rightarrow \alpha \cdot X\beta Y]$ where the suffix βY generates at least one string whose length is shorter than k. (Note that βY may

```
function FOLLOW_STEP(stack, X);
    ts := stack(#stack);
    if #stack = 1 then
        if [ts, X] ∈ VISITED then
            return ∅;
        end if;
        VISITED := VISITED ∪ {[ts, X]};
    end if;
    configs := ∅;
    q := GOTO_0(ts, X);
    for a ∈ DR(ts, X) loop
        configs := configs ∪ {[stack+[q], a]};
    end loop;
    for (q, Y) | (ts, X) reads (q, Y) loop
        configs := configs ∪ FOLLOW_STEP(stack + [q], Y);
    end loop;
    for [C → γ· X] ∈ ts | C ≠ S' loop
        if |γ| + 1 < #stack then
            configs := configs ∪ FOLLOW_STEP(stack(1..#stack - |γ|), C);
        else
            assert γ = γ1γ2 where |γ2| + 1 = #stack;
            for q ∈ PRED(stack(1), γ1) loop
                configs := configs ∪ FOLLOW_STEP([q], C);
            end loop;
        end if;
    end loop;
    return configs;
end FOLLOW_STEP;
```

Figure 3.13: FOLLOW$_k$ *function*

---

actually be ϵ.) In such a case, $stack = [p, p_1, \ldots, p_n]$, where $|X\beta| = n$ and the sequence of states $p_1, \ldots, p_n$ are the states traversed by the automaton when executing $\mathrm{GOTO}_0(p, X\beta)$. Thus, knowing the relevant items $[C \rightarrow \alpha X\beta \cdot Y] \in p_n$ that cannot be extended, the source items $[C \rightarrow \alpha \cdot X\beta Y]$ can be obtained by simply moving the "dot" back n symbols.

From Lemma 3.1.3, one observes that the computation of $\mathrm{FOLLOW}_k(p, A)$ is based on the sets $\mathrm{READ}_k(p, A)$ and $\mathrm{SHORT}_k(p, A)$. Each element $w \in \mathrm{READ}_k(p, A)$, is added directly to $\mathrm{FOLLOW}_k(p, A)$ and each element of $\mathrm{SHORT}_k(p, A)$ is extended into a set of strings of length k which are then added to $FOLLOW_k(p, A)$. To extend a short string $w \in \mathrm{SHORT}_k(p, A)$, the items whose suffixes produced w must be identified and each string of length $k - |w|$ that can *follow* the left-hand side of such items, in the context of the state p, is appended to w. Unfortunately, these equations cannot be used in a straightforward manner to derive an algorithm to compute FOLLOW_k sets. Once again, one has to avoid getting into an infinite loop. Just as in the case of $k = 1$, this may occur when the **includes** relation contains one or more cycles. However one cannot use the digraph algorithm as was done in the case of FOLLOW_1 sets because the equations Lemma 3.1.3 are not of

the suitable form for that algorithm. Hence, a recursive algorithm must be used and this algorithm must keep track of all state-nonterminal pairs that are visited.

Combining the READ_{k*} sets above with the equation of the Lemma 3.1.3, one obtains the following extended equations for FOLLOW_k sets:

Lemma 3.2.1

$\text{FOLLOW}_0(stack, X) = \{\varepsilon\}$

$\text{FOLLOW}_k(stack, X) =$

$\quad \bigcup\{\{a\}.\text{FOLLOW}_{(k-1)}(stack + [q], a) \mid a \in \text{DR}(ts, X),\ q = \text{GOTO}_0(ts, X)\}$

$\cup \bigcup\{\text{FOLLOW}_k(stack + [q], Y) \mid (ts, X)\ \textbf{reads}\ (q, Y)\}$

$\cup \bigcup\{\text{FOLLOW}_k(stack(1..\#stack - |\gamma|), C) \mid [C \rightarrow \gamma \cdot X] \in ts,\ |\gamma| + 1 < \#stack\}$

$\cup \bigcup\{\text{FOLLOW}_k([q], C) \mid [C \rightarrow \gamma_1\gamma_2 \cdot X] \in ts,\ |\gamma_2| + 1 = \#stack,\ q \in \text{PRED}(stack(1), \gamma_1)\}$.

These equations are said to yield the set of symbols of length k that can follow in the context of the configuration $(stack, X)$. The configuration $([p], A)$ is used to initiate the computation of $\text{FOLLOW}_k(p, A)$ using the above equations. The function FOLLOW_STEP of Figure 3.13 computes a step for a FOLLOW_k set, based on the equations of Lemma 3.2.1, in the same way as the READ_STEP function of Figure 3.12 computes a step for READ_{k*} set. FOLLOW_STEP differs from READ_STEP in that, given a pair $(stack, X)$ as input, it computes all succeeding terminal configurations of that pair, even those that are outside the context. In other words, FOLLOW_STEP may automatically trigger the computation of other FOLLOW_i sets, where $0 < i <= k$. Prior to any invocation of the FOLLOW_STEP function, a global set VISITED must be initialized to the empty set. VISITED is subsequently used in FOLLOW_STEP to keep track of all state-nonterminal pairs that have been seen. A FOLLOW_* function can be designed that uses the FOLLOW_STEP function to compute FOLLOW_k sets just like the READ_* function of Figure 3.12 computes READ_{k*} sets using the READ_STEP function.

From Lemma 3.1.2, one observes that full LALR_k sets for a given state-item pair can be easily computed given the relevant FOLLOW_k sets. However, the computation of full LALR_k sets (or full FOLLOW_k sets for that matter) is not of primary interest. In the next section, the incremental computation of minimal LALR_k lookahead sets for a given state is explored.

3.3 Computation of Varying-Length Lookahead Strings

As described earlier, conflicts are resolved one at a time, in each inconsistent state, for each conflict symbol. Recall that an LR(0) state is inconsistent if it consists of two or more items and at least one of these items is a final item. In such a case, one cannot determine which action to execute without knowing which input symbol or symbols can be expected. Thus, this initial inconsistency of q can be viewed as a conflict on ϵ. An attempt is made to resolve it by computing the set of LALR(1) lookahead sets that can appear after each final item. In other words, without consuming any input symbol, these final items can cause a reduction that takes the automaton into other states where these lookahead symbols can be read. Since $\text{GOTO}_0(q, \epsilon) = q$, state q can also be reached without any consumption of input symbols. The actions that can be executed without consuming any input symbol will be referred to as ϵ-actions.

A state q is said to be LALR(1), if the intersection of its LALR(1) lookahead sets and the set of terminals on which shift actions are defined in it is empty. If a state q is not LALR(1), each symbol a in this intersection is an LALR(1) conflict symbol that must be extended into a set of lookahead strings of length 2 in an attempt to disambiguate q. If conflict symbols are detected in the sets of symbols that were appended to a to extend it into a set of lookahead strings of length 2, the process is repeated for these LALR(2) conflict symbols, then for LALR(3) conflict symbols if any, and so on, up to $k-1$.

In order to extend an LALR(1) conflict symbol a in a state q into a longer lookahead string, the relevant actions with which a are associated must be identified. This will include one or more reduce actions and perhaps a shift action. Once the relevant actions have been identified, one must find all the *sources* where the symbol a can be read for each action. For a shift action, the source is the state q itself. For a reduce action, the sources can be found by retracing the paths of the automaton used to compute the LALR(1) lookahead sets, looking for the relevant states where the symbol a can be read. It is not hard to see that the set of sources for the reduce action of a final item $[A \rightarrow \omega\cdot]$ in a state q is the following set:

$$stacks = \{stack \mid [stack, a] \in \text{FOLLOW_STEP}([p], A),\ p \in \text{PRED}(q, \omega)\}.$$

Knowing the set of sources *stacks* of a symbol a that is associated with a given item, the symbol can be extended into a set of lookahead strings of length 2 as follows:

$$\{a.b \mid [stack, b] \in \text{FOLLOW_STEP}(stk, A),\ stk \in stacks\}.$$

Furthermore, note that as the FOLLOW_STEP function is extending the lookahead, it is also computing the next set of sources. Hence, the FOLLOW_STEP function can be adapted to help compute the minimal LALR(k) lookahead sets. However, one can do better by splitting this function into two special purpose functions: FOLLOW_SOURCES (Figure 3.14), which given a configuration $(stack, X)$ and a terminal a returns the set of all possible sources of a in the context of $(stack, X)$; and a function NEXT_LA (Figure 3.15), which given a configuration $(stack, X)$ returns the set of terminals that can follow in the context of a configuration $(stack, X)$. Note that in NEXT_LA, instead of retracing the **reads** and **includes** paths each time, the READ_1 and FOLLOW_1 sets are used since they are already available for the computation of LALR(1) lookahead sets.

With these two functions as building blocks, the LOOK_AHEAD function of Figure 3.16 computes the minimal LALR(k) lookahead sets with varying-length string in a straightforward manner. LOOK_AHEAD takes as argument an inconsistent state q. Its first step is to initialize the ACTION map for each pair (q, a), $a \in T$, into the singleton {shift p}, if $\text{GOTO}_0(q, a) = p$, or into the empty set, otherwise. (Initially, ACTION = $\emptyset$.) Next, LOOK_AHEAD computes the LALR(1) lookahead set for each final item in q and uses the symbols in these lookahead sets to update the ACTION map with the relevant reduce actions. After having completed this process, if q is a LALR(1) state, #ACTION$(q, a) <= 1$, $\forall a \in T$. Otherwise, each terminal that does not satisfy this condition is an LALR(1) conflict symbol. For each such symbol a, the next step of LOOK_AHEAD is to compute the sources of a for each action with which a is associated and invoke the procedure RESOLVE_CONFLICTS of Figure 3.17 to extend a into an appropriate set of non-conflicting lookahead strings.

```
function FOLLOW_SOURCES(stack, X, a);
    ts := stack(#stack);
    if #stack = 1 then
        if [ts, X] ∈ VISITED then
            return ∅;
        end if;
        VISITED := VISITED ∪ {[ts, X]};
    end if;
    q := GOTO_0(ts, X);
    if a ∈ DR(ts, X) then
         stacks := {stack+[q]};
    else stacks := ∅;
    end if;
    for (q, Y) | (ts, X) reads (q, Y) loop
        stacks := stacks ∪ FOLLOW_SOURCES(stack + [q], Y, a);
    end loop;
    for [C → γ·X] ∈ ts | C ≠ S' loop
        if |γ| + 1 < #stack then
            stacks := stacks ∪ FOLLOW_SOURCES(stack(1..#stack - |γ|), C, a)}
        else
            assert γ = γ1γ2 where |γ2| + 1 = #stack;
            for q ∈ PRED(stack(1), γ1) loop
                stacks := stacks ∪ FOLLOW_SOURCES([q], C, a);
            end loop;
        end if;
    end loop;
    return stacks;
end FOLLOW_SOURCES;
```

Figure 3.14: FOLLOW_SOURCES *function*

RESOLVE_CONFLICTS is a recursive procedure that is invoked with 4 arguments: an inconsistent state q; a conflict symbol t; a mapping sources that takes each action act in the set ACTION(q, t) into the set of sources where t can be read following a sequence of ϵ-actions induced by act in q; and an integer n that indicates that t is an LALR(n) conflict symbol. RESOLVE_CONFLICTS first checks whether or not $n > k$. If so, it returns immediately. Otherwise, it allocates a lookahead state p, initializes ACTION$(p, a) = \emptyset$, $\forall a \in T$ and resets ACTION(q, t) to a single lookahead shift action to the new state p. This is because when in state q, if the next input symbol is t, the parser needs to perform more lookahead actions in order to determine its next move. Next, a new set of lookahead symbols is computed for each action and for each new lookahead symbol a, ACTION(p, a) is updated accordingly. If no conflicts are detected in the new lookahead sets, the original LR(0) state that started this process is LALR(n). Otherwise, for each LALR(n) conflict symbol, new sources are computed and the process is repeated with a recursive invocation of RESOLVE_CONFLICTS.

After the LOOK_AHEAD function has been invoked to resolve conflicts for each incon-

```
function NEXT_LA(stack, X);
    ts := stack(#stack);
    q := GOTO_0(ts, X);
    la := READ_1(ts, X);
    for [C → γ·Xδ] ∈ ts | δ ⇒* ε and C ≠ S' loop
        if |γ| + 1 < #stack then
            la := la ∪ NEXT_LA(stack(1..#stack - |γ|), C);
        else
            assert γ = γ1γ2 where |γ2| + 1 = #stack;
            for q ∈ PRED(stack(1), γ1) loop
                la := la ∪ FOLLOW_1(q, C);
            end loop;
        end if;
    end loop;
    return la;
end NEXT_LA;
```

Figure 3.15: NEXT_LA *function*

---

sistent LR(0) state, if $k > 0$, the grammar is LALR(k) if and only if the following three conditions are satisfied:

1. the **reads** relation contains no cycle;
2. $\forall A \in N,\ A \not\Rightarrow^*_{rm} A$;
3. for each LR(0) or lookahead state p and terminal symbol a, $\#\mathrm{ACTION}(p, a) \leq 1$.

If either condition 1 or 2 is not satisfied, the grammar is not LR(k) for any k. If only condition 3 is not satisfied, it may still be possible to construct a deterministic LALR parser for the grammar by increasing the value of k. However, it may also be that the grammar is ambiguous or that it is an LR(k) grammar. In those cases, no amount of extra lookahead will help.

3.4 Remarks

In practice, this incremental approach for constructing LALR(k) parsers is very efficient because most constructs in an LALR(k) grammar are in fact LR(0) or LALR(1). Moreover, when it is necessary to use more lookahead, an LALR(k) parser with varying-length lookahead strings is often as space-efficient as an LALR(1) parser that is obtained by transforming an LALR(k) input grammar. This is because duplication of some constructs is usually necessary to render an LALR(k) grammar LALR(1). Thus, the resulting LALR(1) parsing tables may contain more actions than their LALR(k) counterpart for the original grammar. For example, consider the Pascal grammar in Appendix F. That grammar is LALR(2) because of the following constructs:

```
procedure LOOK_AHEAD(q)
    for a ∈ T loop
        ACTION(q, a) := if (p := GOTO_0(q,a)) ≠ Ω then {shift p} else ∅ end;
    end loop;
    for each final item A → ω· ∈ q loop
        for a ∈ LA(q, A → ω·) loop
            ACTION(q, a) := ACTION(q, a) ∪ {reduce A → ω};
        end loop;
    end loop;
    for a ∈ T | #ACTION(q, a) > 1 loop
        sources := ∅;
        for act ∈ ACTION(q, a) loop
            if act is a shift action then
                sources(act) := {[q]};
            else
                assert act = reduce A → ω;
                sources(act) := ∅;
                for p ∈ PRED(q, ω) loop
                    VISITED := ∅;
                    sources(act) := sources(act) ∪ FOLLOW_SOURCES([p],A,a);
                end loop;
            end if;
        end loop;
        RESOLVE_CONFLICTS(q, a, sources, 2);
    end loop;
    return;
end LOOK_AHEAD;
```

Figure 3.16: Procedure to compute lookahead sets with variable-length strings

---

```
if_statement    →  IF expression THEN
                       statement
                |  IF expression THEN
                       restricted_statement [;]
                   ELSE statement
[;]             →  ϵ
                |  ;

variant_part    →  CASE tag_field type_identifier OF variant_list
tag_field       →  ϵ
                |  field_identifier :
field_identifier →  IDENTIFIER
type_identifier  →  IDENTIFIER
```

An optional ";" was added before ELSE in the if_statement rule above. This is an extension of the syntax of Pascal. Since Pascal programmers commonly make the mistake of inserting an extraneous semicolon in front of ELSE, it is preferable to have the parser be able to accept such an input and emit a "warning" message for it instead of an error. Unfor-

```
function RESOLVE_CONFLICTS(q, t, sources, n);
    if n > k then
        return;
    end if;
    allocate a new state p;
    for a ∈ T loop
        ACTION(p, a) := ∅;
    end loop;
    ACTION(q, t) := {la-shift p};
    for stacks ∈ sources(act) loop
        for stk ∈ stacks loop
            for a ∈ NEXT_LA(stk, t) loop
                ACTION(p, a) := ACTION(p, a) ∪ { act };
            end loop;
        end loop;
    end loop;
    for a ∈ T | #ACTION(p, a) > 1 loop
        new_sources := ∅;
        for act ∈ ACTION(p, a) loop
            new_sources(act) := ∅;
            for stk ∈ sources(act) loop
                VISITED := ∅;
                new_sources(act) := new_sources(act) ∪ FOLLOW_SOURCES(stk,t,a);
            end loop;
        end loop;
        RESOLVE_CONFLICTS(p, a, new_sources, n+1);
    end;
    return;
end RESOLVE_CONFLICTS;
```

Figure 3.17: RESOLVE_CONFLICTS procedure

tunately, when ";" is used in this context, the parser does not know whether to reduce the handle by statement or restricted_statement. If, on the other hand, the parser can consult two symbols, the string "; ELSE" instructs it to reduce the handle to restricted_statement. This LALR(2) conflict cannot easily be removed from the grammar without some major restructuring of the rules. (But, since it is an extension it will not be discussed any further.)

In the case of a variant_part, after shifting the symbol CASE, if the parser can only consult one input symbol and that symbol is IDENTIFIER, the parser does not know whether to shift it, because it is a field_identifier; or to reduce tag_field by the empty rule, because it is a type_identifier. This conflict can be resolved by duplicating the variant_part rule and removing the ϵ choice for tag_field as follows:

```
variant_part  →  CASE type_identifier OF variant_list
              |  CASE tag_field type_identifier OF variant_list
tag_field     →  field_identifier :
```

Table D.1 in Appendix D gives some information about different LALR parsers constructed with the method described in this thesis. In that table, Pascal2 refers to the LALR(2) grammar of Appendix F; Pascal1 is the same grammar as Pascal2, but without the optional semicolon preceding ELSE in the if_statement; Pascal is the same grammar as Pascal1, but with the variant_part and tag_field rules modified as suggested above. Note that the parsing tables for Pascal1 contain fewer actions and are slightly smaller than the parsing tables for Pascal.