A Practical method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery

Philippe Charles
May 1991

A dissertation in the Department of Computer Science
submitted to the faculty of the Graduate
School of Arts and Science in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
at New York University.

Edmond Schonberg, Research Advisor

ABSTRACT

LR parsing is used for a wide range of applications, including compiler construction, automatic code generation, language-specific editors and natural language processing. Currently, however, solutions have not been developed for practical multiple-lookahead parsing, fully-automatic error recovery, and space and time-efficient LR parsing across the wide-range of applications.

A practical framework for LR(k) parsing is introduced. An efficient algorithm incrementally constructs an LALR(k) parser with varying-length lookahead strings, whose symbols are consulted during parsing only when necessary. Currently, effective LR error recovery systems require some user intervention. An effective and fully automated syntactic error recovery method for LR(k) parsers is presented. A generally effective method for compressing LR(k) parsing tables is also presented.

These innovations have been incorporated into a parser generator system which automatically produces a production-quality parser with error diagnostics and recovery.

CONTENTS

1. LALR(K) PARSING

1.1 Introduction

In 1965, Knuth [1] introduced LR(k) parsing, a bottom-up syntax analysis technique that can be used to recognize the largest class of deterministic context-free languages. (The "L" stands for left-to-right scanning of the input, the "R" is for constructing a rightmost derivation in reverse, and the k is for the number of input symbols of lookahead that are used in making parsing decisions.) Over the years, this parsing method has attracted much attention because in addition to its ability to recognize a large class of languages the resulting parsers offer the following advantages:

- they can be constructed automatically from a context-free grammar definition;
- they are time-efficient since they can accept or reject an input in a single left-to-right scan of it with no backup;
- they can detect an error at the earliest possible point.

A context-free grammar is said to be LR(k) if an LR(k) parser can be successfully constructed from it. A language is said to be LR(k) if it can be defined by an LR(k) grammar. In their canonical form, LR(k) parsers (when $k > 0$) usually require too much space to be of practical use. (The relationship between an arbitrary context-free grammar and the size of its canonical LR(k) parser has never been precisely demonstrated, but for a typical programming language grammar, when $k = 0$ the parser usually contains several hundred states; when $k = 1$, parsers with several thousand states are common.) As a result, two variants of LR(k) parsers which were invented by DeRemer have gained popularity over the years. They are known as LookAhead LR(k) (LALR(k)) introduced in 1969 and described in [3] and simple LR(k) (SLR(k)) introduced in 1971 and described in [4]. These variants of LR(k) parsers are relatively space-efficient because their underlying automaton is the LR(0) machine, regardless of the value of k. The set of languages that is SLR(k) is a proper subset of the set of LALR(k) languages which, in turn, is a proper subset of the set of LR(k) languages. However, in practice, LALR(k) grammars are used because they are sufficiently powerful to accomodate most programming language constructs.

By keeping the number of states at a minimum, the SLR(k) and LALR(k) variants help reduce the space requirement of an LR(k) parser while retaining the speed advantage of the latter.

The symbols of a context-free grammar are divided into two classes: terminals (input symbols) and nonterminals (phrase symbols). An LR(k) parser (or variant) for a context-free grammar is a deterministic pushdown automaton that can be represented by two matrices: ACTION, which represents the mapping of a parsing action function and GOTO, which represents the mapping of a goto function. These matrices will be referred to generically as *parsing tables.*

The parsing action function takes as arguments a state and a string of k terminals (called lookahead string) and produces one of four values: **shift**, **reduce** i, **error** or **accept**. The goto function is the transition function of the automaton. It takes as arguments a state and a grammar symbol (terminal or nonterminal) and produces either a new state that the parser should enter or **error**. Thus, the rows of the parsing tables are indexable by a state of the automaton, each column of ACTION is indexable by a string of terminals of length k and each column of GOTO is indexable by a distinct grammar symbol. Each entry in a parsing table is either a *useful entry* that represents a valid move to be taken by the automaton (for the corresponding pair of indices) or an *error entry.*

1.2 The Problems

As can be observed from the definition of the parsing tables, for a given grammar, the number of states (rows) in its SLR(k) or LALR(k) automaton and the number of columns in its GOTO matrix remain fixed for any value of k; but, the number of columns in its ACTION matrix is exponential with respect to k. However, if the error entries are kept as empty slots these matrices are very sparse. Typically, less than 2% of the entries in the parsing tables of an LALR(1) parser are useful.

One of the most important issues in LR parsing is to find suitable data structures for these parsing tables whose space requirement is, at worst, proportional to the number of useful entries in the tables but whose time-efficiency is comparable to that of the matrix representation. Another important issue is that of providing an efficient error recovery system for this parsing framework. In particular, the LR(k) variants lose the inherent ability of their canonical counterpart to detect an error at the earliest possible point.

The attempt to make LR(k) parsers more useful, that is, faster, smaller and more automated raises the following important questions:

- How can such a parser be constructed efficiently?
- How can error recovery be done in this framework? More specifically, can an automatic or semi-automatic error recovery system be constructed that works with all LR(k) parsers?
- What is the relationship between the specific type of parsing and the speed and size of the resulting automaton? (For example, the use of extra lookahead symbols can potentially affect both size and speed).
- How can the parse tables be represented compactly without sacrificing speed?

Several papers have been published that address these problems. In the following subsections, the results of some of the seminal work in these areas are briefly described followed by a description of the major innovations of this thesis.

1.2.1 LALR(k) Parser Construction

LALR(k) parsers are almost always used because they are more space-efficient than both the LR(k) and SLR(k) parsers and, in addition, they are more powerful than the SLR(k) parsers. Most commercially available parser generators only deal with the case of $k = 1$.

An LALR(k) parser can be constructed by first building an LR(k) parser and then merging some states. However, this approach is impractical since it is usually difficult to construct the LR(k) parser because of its large space requirement. Instead, a two-step approach is usually taken. In the first step, an LR(0) automaton is constructed (all LALR(k) parsers are based on this automaton); and in the second step the resulting tables are augmented with the necessary lookahead actions.

Informally, an LR(0) item is a context-free grammar rule with a marker that separates its right-hand side into a prefix that has been processed and a suffix that has yet to be processed. An item is called final when the marker indicates that its prefix consists of the whole right-hand side (and its suffix is empty). Each state of an LR(0) automaton corresponds to a set of items.

The LALR(k) lookahead set for a final item in a state of an LR(0) automaton is a set of terminal strings of length k. During parsing, if the next k symbols in the input matches one of the lookahead strings, the parser should perform a reduction by the specific rule from which the relevant item is derived. Hence, in the second step of the construction of an LALR(k) parser, for each state and each lookahead string element of a lookahead set computed in that state, the corresponding ACTION matrix entry is updated to indicate the relevant reduce action. Thus, the main issue in constructing an LALR(k) parser is: *how can the lookahead sets be computed efficiently with respect to both space and time?*

DeRemer presented an algorithm for constructing SLR(k) parsers [4]. However, an algorithm to compute lookahead sets for an LALR parser directly on the LR(0) automaton (even for the case $k = 1$) had remained elusive until 1971, when Lalonde [6] presented his algorithm. Other algorithms for computing LALR(1) lookahead sets on the LR(0) automaton were published throughout the 1970's, including the method used in Yacc [10] which is also described in detail in [32], and the lane-tracing method of Pager [13]. These algorithms were not very efficient.

Much progress was made in the area of computing LALR(1) lookahead sets in the 1980's. The algorithm of DeRemer and Penello, published in 1982 [26], still remains the best, especially when it is implemented with some improvements suggested in [25]. Bermudez and Logothethis, in 1989 [38], proposed a theoretically interesting approach that reduces the problem of computing LAFOLLOW sets, which are used to compute LALR lookahead sets, to the problem of computing regular FOLLOW sets, which are used to construct SLR lookahead sets. The time complexity of this algorithm is potentially the same as that of the DeRemer and Penello algorithm, as will be shown later, but the space requirement is greater. Other algorithms have also been published, most notably: Kristensen and Madsen, 1981 [24]; Park, Choe and Chang, 1985 [31]; Ives, 1986 [35]. All of these algorithms, however, are less efficient than the DeRemer and Penello approach.

Only Kristensen and Madsen generalized their algorithm to compute lookahead sets for $k > 1$. However, their generalized algorithm is only of theoretical interest, in that it computes the complete lookahead sets required for each ambiguous state. As can be observed from the definition of the ACTION matrix, the problem of computing the full lookahead sets for an LALR(k) parser is inherently intractable in that the size of the solution itself may be exponential with respect to k.

1.2.2 Error recovery

Error recovery is traditionally divided into simple recovery [21] [36], phrase-level (or secondary) recovery [9] [21] [27] [36], and scope recovery [36].

In simple recovery, an attempt is made to repair an erroneous input by using primitive editing operations on the error symbol. That is, a symbol may be inserted before it, it may be replaced by another symbol, or it may be deleted.

In phrase-level recovery, a sequence of zero or more tokens in the vicinity of the error symbol is discarded from the input or replaced by a nonterminal. The *error productions* approach of Yacc is a form of secondary recovery where the nonterminal candidates to be chosen for this kind of repair are identified by productions whose right-hand sides include a special terminal symbol called the *error symbol.* Sippu and Soisalon-Soininen [27] presented a more sophisticated method for secondary recovery which does not require the use of error productions but is somewhat expensive in that it requires that some information be computed at run-time.

Scope recovery was introduced by Burke and Fisher. The idea is to insert a sequence of closing syntactic fragments into the text, where appropriate, to complete the specification of certain blocks or block-like structures. This recovery approach is very effective when used in conjunction with primary and secondary recovery as suggested in [36]. However, each relevant closing fragment had to be specified explicitly as a sequence of terminal symbols. Therefore, their method required that a user be familiar with the language in question.

The error recovery method of Burke and Fisher is the most practical and effective method to date. However, it is based on a deferred parsing technique which requires a double parsing of the input even for correct programs. In addition to the introduction of scope recovery, Burke and Fisher also made some improvements in primary recovery by considering *merging* of two adjacent tokens and *misspelling* of keywords. Other error recovery methods (e.g. [17]) have been published, but they are mostly of theoretical interest and are not used in practice.

1.2.3 Parsing Tables

The issue of LR parsing table compression has been widely studied, but up to now, no general method has been produced that is well-suited across the range of different applications. Table compression is still treated in the literature as a time versus space problem. Depending on the application, techniques from sparse matrix representation with sequential searching, hashing, and other more time-efficient but space-consuming direct access methods have been proposed.

The table compression technique used in Yacc [9] consists of a combination of direct access techniques for transitions and sequential search techniques for reduce actions. For small grammars, this is an acceptable approach. However, this approach does not always perform well on large grammars. Tarjan and Yao [23] published an analysis of the direct access method of Ziegler, and formulated the precise conditions under which that compression technique performs well. In 1984, Dencker, Durre and Heuft [30] presented a direct access technique based on graph coloring which does very well in minimizing space. Unfortunately, their approach requires referencing a packed boolean matrix to test the validity of each action. In practice, this test renders their method slower than the Yacc method.

1.3 Contributions of the Thesis

In this thesis, several contributions are made in each of the areas mentioned above. These results have been integrated in a parser generator system that automatically produces efficient LALR(k) parsers with error recovery from a context-free grammar definition. These innovations are summarized as follows:

- A new framework for LALR(k) parsers. As pointed out before, the construction of a traditional LALR(k) parser is impractical since the size of the lookahead sets required for such a parser can be exponential. The approach taken in this method can be best described as generating an LALR(k) parser with *variable-length lookahead strings.* A lookahead set for a final item in a given state of an LR(k) parser consists of the set of strings of length k that may appear in the input when the parser enters that state. In an LALR(k) parser with variable-length lookahead strings, each lookahead set is replaced by a minimum subset of prefixes of its string elements that is sufficient to render the parser deterministic. This new framework is discussed in chapter 3.

- A practical algorithm for constructing variable LALR(k) parsers. This method not only computes the minimum amount of lookahead information required but it does it in an incremental fashion. Thus, the space needed to construct these sets is kept to a minimum. This algorithm is presented in section 3.3.

- A fully automatic error recovery method which is more practical and efficient than other known methods. This language- and machine-independent method is applicable to all forms of LR(k) parsing but it is especially effective in the context of a parser generated by the above method. Error Recovery is the subject of chapter 4.

- A practical and effective method for compressing LR(k) parsing tables. This compression method is also applicable to all forms of LR(k) parsers, but it is particularly effective in this framework. Table compression is covered in chapter 5.

2. THE PARSER GENERATOR

2.1 Basic Concepts and Terminology

A context-free grammar (CFG) is a quadruple (N, T, P, S), where N is a finite set of nonterminal symbols, T is a finite set of terminal symbols distinct from N, S is a distinguished symbol of N called the start symbol, and P is a finite set of productions, each of the form $A \rightarrow \omega$, where $A \in N$ and $\omega \in V^*$. Given a grammar G, V (the vocabulary) stands for $N \cup T$.

Lower-case Greek letters such as α, β and γ are used to denote strings in V^*. Lower-case Roman letters at the beginning of the alphabet (a, b, c) and t are used to denote symbols in T while those near the end of the alphabet (x,y,z) denote strings in T^*. Upper-case letters near the beginning of the alphabet (A,B,C) denote nonterminals in N while those near the end (X,Y,Z) denote symbols in V. The empty symbol is denoted ϵ and the empty string is denoted ε. The end-of-file token is denoted by $\perp$. The length of a string γ is denoted $|\gamma|$.

The following SETL2 [41] notation will also be used. The symbol Ω denotes the special "undefined value" constant. A finite ordered sequence of arbitrary elements, called a *tuple*, will be denoted by listing the elements in the correct order, within the brackets '[' and ']'. If T is a tuple, $T(i)$ is the ith element of T and $T(m..n)$ is the tuple consisting of the elements $T(m), T(m+1), \ldots, T(n)$, if $m >= n$ and the empty tuple, $[\,]$, otherwise. If T_1 and T_2 are tuples, then $T_1 + T_2$ is the tuple obtained by appending the sequence of elements in T_2 at the end of the sequence of elements of T_1. A single-valued map from a finite set A (the domain) to a finite set B (the range) will be represented as a set of ordered pairs $[x, y]$, where $x \in A$, $y \in B$ and each element of A is mapped to at most one element of B. Given a map M, and an element x in its domain, $M(x)$ represents the element y in the range of M that is paired with x (y is called the *image* of x). If X is a tuple, set or map, its length or cardinality is denoted $\#X$.

From now on it is assumed that a given grammar G has been augmented with a new starting rule $S' \rightarrow S\perp^k$ and G contains no *useless* nonterminals. A nonterminal A is said to be useless if it does not generate any string of terminals; i.e., $A \not\Rightarrow^+ w$ for any $w \in T^*$.

For a given context-free grammar,

$$\text{FIRST}_k(\alpha) = \{x \mid (\alpha \Rightarrow^*_{lm} x\beta \text{ and } |x| = k) \text{ or } (\alpha \Rightarrow^* x \text{ and } |x| < k)\}.$$

That is, $\text{FIRST}_k(\alpha)$ consists of all terminal prefixes of length k (or less if α derives a terminal string of length less than k) of the terminal strings than can be derived from α. Closely related to the FIRST_k function is the *ε-free first* function, $\text{EFF}_k(\alpha)$, which is defined as all the elements of $\text{FIRST}_k(\alpha)$ whose derivation does not involve replacing a leading nonterminal by ε. More formally,

$$\mathrm{EFF}_k(\alpha) = \{w \mid \alpha \Rightarrow^*_{rm} \beta \Rightarrow^*_{rm} wx,\ \beta \neq Awx\ \forall A \in N \text{ and } \{w\} = \mathrm{FIRST}_k(wx)\}$$

.

If x and y denote arbitrary strings then $x.y$ is the string obtained by concatenating the string denoted by y to the string denoted by x. Let M and N be two sets of strings, the concatenation operation is extended to sets of strings as follows:

$$M.N = \{x.y \mid x \in M,\ y \in N\}$$

If $M, N \subseteq T^*$ then

$$M \oplus_k N = \bigcup\{\mathrm{FIRST}_k(w) \mid w \in M.N\})$$

2.1.1 LR(k) parsers

An LR(k) item is a quadruple (A, α, β, u), written $[A \to \alpha \cdot \beta, u]$, where $A \to \alpha\beta \in P$ and $u \in T^k$ is a lookahead set. A is called the *left side*, α is called the *prefix*, β is called the *suffix*. The first symbol in β, immediately following the dot, is called the *dot symbol.* When $\beta = \varepsilon$, the item is called a *final item* and the dot symbol is considered to be ϵ.

Let K be a set of LR(k) items. The *closure* of K, denoted $CLOSURE(K)$ is defined as the smallest set satisfying the equation:

$$\begin{aligned}\mathrm{CLOSURE}(K) = K \ \cup\ \{[B \to \cdot\gamma, v] \mid v \in \mathrm{FIRST}_k(\beta u),\\ [A \to \alpha \cdot B\beta, u] \in \mathrm{CLOSURE}(K),\ B \to \gamma \in P\}\end{aligned}$$

Let p be a closure set. The *kernel* set of p, denoted KERNEL(p) is the smallest subset of the LR(k) items in p such that:

$$p = \mathrm{CLOSURE}(\mathrm{KERNEL}(p))$$

Given a set of items, p, for each dot symbol X that appears in an item of p, a *goto function*: GOTO$_k$, is defined on the pair (p, X) as follows:

$$\mathrm{GOTO}_k(p, X) = \mathrm{CLOSURE}(\{[A \to \alpha X \cdot \beta, u] \mid [A \to \alpha \cdot X\beta, u] \in p\})$$

For a given grammar $G = (N, T, P, S)$, the canonical set of LR(k) items for G, denoted I_k^G, can be constructed with the following procedure given a *closure function* (to compute CLOSURE(K) for some set of items K) and GOTO$_k^G$.

1. Initialize $I_k^G = \emptyset$

2. Start with a kernel set consisting solely of the *initial item*: $[S' \to \cdot S]$; compute its closure set and add that closure set to I_k^G.

3. Chose a closure set p from I_k^G. Compute its set of dot symbols and apply the GOTO$_k$ function on p and each of its dot symbols. If any new closure sets not yet in I_k^G are thus obtained they are added to I_k^G.

4. Repeat the preceeding step until no more new closure sets can be added to I_k^G.

This algorithm must clearly terminate, since the set of items and the set of symbols are finite.

Definition 2.1.1: Let G be a context-free grammar. The LR(k) machine for G is a triple: $LRM_k^G = (M_k^G, \mathrm{IS}_k^G, \mathrm{GOTO}_k^G)$, where M_k^G is a set of LR(k) states, one for each set of items in I_k^G. IS_k^G is the initial state corresponding to the closure set of the initial item. GOTO_k^G is the GOTO function defined on $M_k^G \times V \to M_k^G$.

Observe that a state p in M^k is characterised by its *kernel* set since the complete set of items making up that state can be reproduced given the kernel set and the closure function. For convenience, no distinction will be made, from now on, between a state and its corresponding set of items. Also, for a given grammar G, the superscript G will be omitted whenever this omission causes no confusion. An item in p that is in KERNEL(p) is called a kernel item of p. An item in p that is not in KERNEL(p) is called a *closure* item.

It is also convenient to generalize the GOTO_k function for arbitrary strings as follows:

$$\begin{aligned}\mathrm{GOTO}_k(p,\varepsilon) &= p\\ \mathrm{GOTO}_k(p,X\alpha) &= \mathrm{GOTO}_k(\mathrm{GOTO}_k(p,X),\alpha)\end{aligned}$$

Let PRED be the inverse of the GOTO_k function. It is defined on arbitrary strings as follows:

$$\mathrm{PRED}(p,\alpha) = \{q \mid \mathrm{GOTO}_k(q,\alpha) = p\}$$

2.1.2 LALR(k) parsers

The notion of LALR(k) parsers is captured by the following definitions and theorems presented in [24]. In each, let G be a CFG with LR(k) states M_k, $k \geq 0$.

Definition 2.1.2: Let $p \in M_k$, then

$$\mathrm{LR}_k(p,[A \to \alpha \cdot \beta]) = \{u \mid [A \to \alpha \cdot \beta, u] \in p\}$$

Definition 2.1.3: Let $[A \to \alpha \cdot \beta, u]$ be an LR(k) item and let $p \in M_k$, then

$$\mathrm{CORE}([A \to \alpha \cdot \beta], u) = [A \to \alpha \cdot \beta]$$

and

$$\mathrm{CORE}(p) = \{\mathrm{CORE}(I) : I \in p\}$$

No distinction is made between the items $[A \to \alpha \cdot \beta, \varepsilon]$ and $[A \to \alpha \cdot \beta]$.

Definition 2.1.4: Let $p \in M_0$, then

$$\mathrm{URCORE}_k(p) = \{q \in M_k \mid \mathrm{CORE}(q) = p\}$$

URCORE relates an LR(0) state p with a set of LR(k) states with the same core. Note that since $\text{CORE}(\text{IS}_0) = \text{CORE}(\text{IS}_k)$, and that $\text{GOTO}_k(p, X)$, for all $k > 0$, depends only on the core of p, then each LR(k) state corresponds to an LR(0) state with the same core. In other words, $\text{URCORE}_k(p) \neq \emptyset$, for all $k > 0$ and $p \in M_0$.

Definition 2.1.5: Let $p \in M_0$, then

$$\text{LALR}_k(p, [A \to \alpha \cdot \beta]) = \bigcup \{\text{LR}_k(q, [A \to \alpha \cdot \beta]) \mid q \in \text{URCORE}_k(p)\}$$

Definition 2.1.6: A grammar G is said to be LALR(k), $k \geq 0$ if for all $p \in M_0$ and for all distinct items $[A \to \alpha \cdot \beta]$ and $[B \to \gamma \cdot]$ in p,

$$\text{EFF}_k(\beta) \oplus_k \text{LALR}_k(p, [A \to \alpha \cdot \beta]) \cap \text{LALR}_k(p, [B \to \gamma \cdot]) = \emptyset$$

An LR(0) machine constructed for a grammar G is in fact a correct parser for G; i.e., the language recognized by LRM_0 is exactly the same language described by G. However, it may be nondeterministic due to the presence of one or more *inconsistent* states. In general, a state is said to be inconsistent if it allows two different moves for a given lookahead string. In particular, a state in M_0 is inconsistent if it contains two or more items and one of these items is a final item.

When a state $p \in M_0$ does not satisfy the condition of definition 2.1.6, it is also said to be *inconsistent* (in an LALR(k) sense). The strings that are in the intersection of the two sets are said to be in conflict and they are called *conflict strings.* If $\beta \neq \varepsilon$ then the resulting conflicts are called *shift-reduce* conflicts, otherwise, they are called *reduce-reduce* conflicts.

Theorem 2.1.1: Let $p \in M_k$, then

$$\text{LR}_k(p, [A \to \alpha \cdot \beta]) = \{w \mid w \in \text{FIRST}_k(y),\ S' \Rightarrow^*_{rm} \gamma A y \Rightarrow \gamma\alpha\beta y,\ \text{GOTO}_k(\text{IS}_k, \gamma\alpha) = p\}$$

Theorem 2.1.2: Let $p \in M_0$, then

$$\begin{aligned}\text{LALR}_k(p, [A \to \alpha \cdot \beta]) = \{w \mid w \in \text{FIRST}_k(y),\ S' \Rightarrow^*_{rm} \gamma A y \Rightarrow \gamma\alpha\beta y,\\ \text{GOTO}_0(\text{IS}_0, \gamma\alpha) = p\}\end{aligned}$$

Theorem 2.1.3: Let $p \in M_k$ then

$$\forall q \in \text{PRED}(p, \alpha) : \text{LR}_k(p, [A \to \alpha \cdot \beta]) = \text{LR}_k(q, [A \to \cdot\alpha\beta])$$

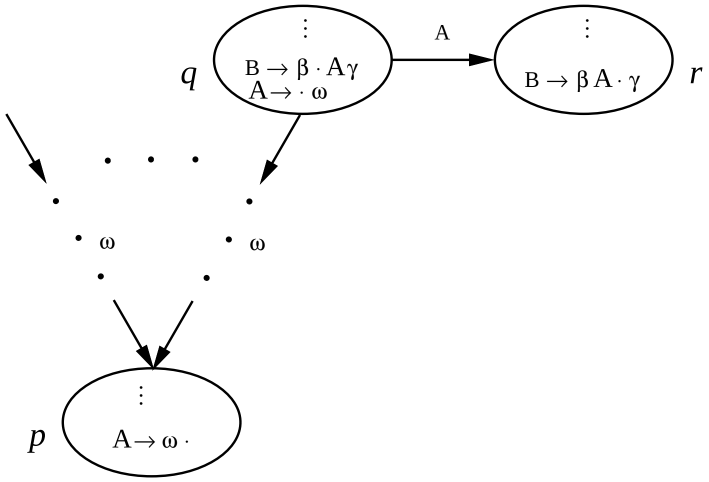


Fig. 2.1: Lookahead paths for final item $[A \to \omega\cdot]$ in state p

Theorem 2.1.4: Let $p \in M_k$ and let $[A \to \cdot\alpha] \neq [S' \to \cdot S\bot^k]$. then

$$\mathrm{LR}_k(p, [A \to \cdot\alpha]) = \bigcup\{\mathrm{FIRST}_k(\gamma) \oplus_k \mathrm{LR}_k(p, [B \to \beta \cdot A\gamma]) \mid [B \to \beta \cdot A\gamma] \in p\}$$

Theorems 2.1.1, 2.1.3 and 2.1.4 follow directly from the algorithm for constructing the canonical collection of LR(k) items. Theorem 2.1.2 may be proved using definition 2.1.5 and theorem 2.1.1.

Note that definition 2.1.5 is a restatement of a point made earlier that an LALR(k) parser can be obtained by first constructing an LR(k) parser and then *merging* certain states of that LR(k) parser. This definition tells us that the relevant LR(k) states that are merged are those having the same core. However, as was also stated earlier, this approach is very inefficient since the space required to construct an LR(k) parser, for $k > 0$ is usually prohibitive. A practical approach to constructing an LALR(k) parser is to first construct the LR(0) machine and then resolve conflicts in inconsistent states of M_0 by computing lookahead strings for final items in such states. The LALR$_k$ lookahead of an item $[A \to \omega\cdot]$ in a state p may informally be described as the set of terminal strings of length k that may appear on input if, during parsing, the reduction $A \to \omega$ can be applied in state p. The objective is to compute $\mathrm{LALR}_k(p, [A \to \omega\cdot])$ using LRM_0. Intuitively, the idea is to simulate all possible steps of the parser following the reduction in order to determine which input strings can be shifted on next.

Consider the set of states where the parsing may resume after a reduction by a rule $A \to \omega$ in a state p; i.e., the set $\mathrm{PRED}(p, \omega)$. Each state $q \in \mathrm{PRED}(p, \omega)$ contains the initial item $[A \to \cdot\omega]$ which was introduced in q through closure by at least one item of the form: $[B \to \beta \cdot A\gamma]$. After reducing by the rule $A \to \omega$ in p, the parser must return to some such state q where a read transition on A places it in a state $r = \mathrm{GOTO}_k(q, A)$. (See Figure 2.1.) The lookahead set $\mathrm{LALR}_k(q, [A \to \cdot\omega])$ consists of all terminal strings of length k that may appear on input after state r is entered. (The validity of this assertion can be confirmed from theorem 2.1.3 and definition 2.1.5.) This set can be divided into two subsets as follows:

$$\begin{aligned}
&\mathrm{LALR}_k(q,[A\to\cdot\omega]) = \\
&\quad\bigcup\{w\in\mathrm{FIRST}_k(\gamma)\mid[B\to\beta\cdot A\gamma]\in q \text{ and } |w|=k\} && (2.1)\\
&\quad\cup\\
&\quad\bigcup\{w\oplus_k\mathrm{LALR}_k(q,[B\to\beta\cdot A\gamma])\mid[B\to\beta\cdot A\gamma]\in q,\ w\in\mathrm{FIRST}_k(\gamma),\ |w|<k\} && (2.2)
\end{aligned}$$

The first subset (2.1) consists of strings of length k that are directly derivable from a suffix γ in a kernel item of r. If, however, $\gamma\Rightarrow^* w$ and $|w|<k$ then the rule $B\to\beta A\gamma$ may reduce before a string of length k is read. Hence, the set of strings following B after such a reduction must be calculated and concatenated to all short strings w. The second subset (2.2) consists of strings that were composed by such concatenations.

Observe that for the case $k=1$ the equation above can be greatly simplified. The function FIRST_1 only yields strings of length 1 and perhaps the empty string. Similarly, the LALR_1 lookahead sets only contain strings of length 1. Therefore, the $\oplus_1$ operator can be replaced by a (conditional) union operator ($\cup$). The equation is rewritten as follows:

$$\begin{aligned}
&\mathrm{LALR}(q,[A\to\cdot\omega]) = \\
&\quad\bigcup\{t\in\mathrm{FIRST}(\gamma)\mid[B\to\beta\cdot A\gamma]\in q\}-\{\varepsilon\} && (2.3)\\
&\quad\cup\\
&\quad\bigcup\{\mathrm{LALR}(q,[B\to\beta\cdot A\gamma])\mid[B\to\beta\cdot A\gamma]\in q,\ \gamma\Rightarrow^*\varepsilon\} && (2.4)
\end{aligned}$$

In [32], the authors refer to the lookahead symbols in the first subset(2.3) of the above equation as *spontaneously* generated lookahead and to the symbols in the second subset(2.4) as propagated lookahead. From now on, when the subscript k is omitted it should be assumed to be 1.

2.2 Previous Work

In this section, some of the most recently published algorithms for constructing LALR parser generators [24] [26] [31] [38] are reviewed. Most of the relevant papers focused only on the case of $k=1$ except [24]. However, that algorithm cannot be implemented in practice, because it requires the computation of complete LALR_k lookahead sets which, as was mentioned earlier, is inherently intractable.

2.2.1 Kristensen and Madsen

Kristensen and Madsen (KM) characterized LALR_k in terms of the LRM_0 machine with the following two lemmas which can be seen as a summary of the somewhat informal discussion at the end of the last section.

Lemma 2.2.1: Let $p\in M_0$, then

$$\mathrm{LALR}_k(p,[A\to\alpha\cdot\beta])=\bigcup\{\mathrm{LALR}_k(q,[A\to\cdot\alpha\beta])\mid q\in\mathrm{PRED}(p,\alpha)\}$$

Lemma 2.2.2: Let $p \in M_0$, $[A \to \cdot\alpha] \in p$, and $A \neq S'$, then

$$\text{LALR}_k(p, [A \to \cdot\alpha]) = \bigcup\{\text{FIRST}_k(\gamma) \oplus_k \text{LALR}_k(p, [B \to \beta \cdot A\gamma]) \mid [B \to \beta \cdot A\gamma] \in p\}$$

The two lemmas above follow directly from theorems 2.1.3 and 2.1.4, and definition 2.1.5. They are combined to obtain the next theorem:

Theorem 2.2.1: Let $p \in M_0$, $[A \to \alpha \cdot \beta] \in p$, and $A \neq S'$, then

$$\begin{aligned}\text{LALR}_k(p, [A \to \alpha \cdot \beta]) = \bigcup\{&\text{FIRST}_k(\delta) \oplus_k \text{LALR}_k(q, [B \to \gamma \cdot A\delta]) \mid \\ &q \in \text{PRED}(p, \alpha) \text{ and } [B \to \gamma \cdot A\delta] \in q\}\end{aligned}$$

Once again, when $k = 1$, the above theorem can be greatly simplified by replacing $\oplus_1$ by a union operator. Kristensen and Madsen also observed that the relevant (non-empty) elements of $\text{FIRST}_1(\delta)$ in the equation of theorem 2.2.1 can be computed directly from the LRM_0 machine. They captured that idea in the following constructive definition and lemma:

Definition 2.2.1: Let $p \in M_0$, then

$$\begin{aligned}\text{TRANS}(p) \quad = \quad & \{a \mid [B \to \beta \cdot a\gamma] \in p\} \ \cup \\ & \bigcup\{\text{TRANS}(\text{GOTO}_0(p, A)) \mid [B \to \beta \cdot A\gamma] \in p \text{ and } A \Rightarrow^* \varepsilon\}\end{aligned}$$

Lemma 2.2.3: Let $p \in M_0$, then

$$\text{TRANS}(p) = \bigcup\{\text{FIRST}_1(\beta) \mid [A \to \alpha \cdot \beta] \in \text{KERNEL}(p)\} - \{\varepsilon\}$$

Using the TRANS sets, theorem 2.2.6 can be reformulated for the case $k = 1$ as follows:

Theorem 2.2.2: Let $p \in M_0$, $[A \to \alpha \cdot \beta] \in p$, and $A \neq S'$, then

$$\text{LALR}_1(p, [A \to \alpha \cdot \beta]) = \bigcup\{L(q, A) \mid q \in \text{PRED}(p, \alpha)\}$$

where

$$\begin{aligned}L(q, A) \quad = \quad & \text{TRANS}(\text{GOTO}_0(q, A)) \ \cup \\ & \bigcup\{\text{LALR}_1(q, [B \to \gamma \cdot A\delta]) \mid [B \to \gamma \cdot A\delta] \in q \text{ and } \delta \Rightarrow^* \varepsilon\}\end{aligned}$$

```
**procedure** TRANS(q)
    TM := TM $\cup$ {q};
    **for** [B $\rightarrow \alpha\cdot$ X $\beta$] $\in$ q **loop**
        **if** X $\in T$ **then**
            LA := LA $\cup$ {X};
        **elseif** X $\Rightarrow^* \varepsilon$ and (GOTO$_0$(q, X) $\notin$ TM) **then**
            TRANS(GOTO$_0$(q, X));
        **end if**;
    **end loop**;
**end** TRANS;

**procedure** LALR(p, [A $\rightarrow \alpha \cdot \beta$]);
    DONE := DONE $\cup$ {(p, [A $\rightarrow \alpha \cdot \beta$])};
    **for** q $\in$ PRED(p, $\alpha$) **loop**
        TM := $\emptyset$;
        TRANS(GOTO$_0$(q, A));
        **for** [B $\rightarrow \gamma\cdot$A$\delta$] $\in$ q | $\delta \Rightarrow^* \varepsilon$ and (q, [B $\rightarrow \gamma\cdot$A$\delta$]) $\notin$ DONE **loop**
            LALR(q, [B $\rightarrow \gamma\cdot$A$\delta$]);
        **end loop**;
    **end loop**;
**end** LALR;

**function** LALR$_1$(p, [A $\rightarrow \alpha \cdot \beta$]);
    DONE := LA := $\emptyset$;
    **if** A = S' **then**
        LA := {$\perp$};
    **else** LALR(p, [A $\rightarrow \alpha \cdot \beta$]);
    **end if**;
    **return** LA;
**end** LALR$_1$;
```

Fig. 2.2: KM LALR$_1$ algorithm

From theorem 2.2.2, a set of equations defining a function LALR_1 from $Item \times State \rightarrow 2^T$ is derived. These equations are then solved in order to compute a LALR(1) lookahead set. The solution of interest is the smallest one satisfying the equations. The algorithm is shown in Figure 2.2.

The function LALR_1 takes as argument an LR(0) item I and a state p and returns the lookahead set for I in p. It invokes two recursive procedures: TRANS and LALR, which are used, respectively, to compute intermediate TRANS and LALR sets. The global set variable LA in used to construct the resulting lookakead set. The other global set variables: DONE and TM are used to prevent LALR and TRANS, respectively, from being visited more than once with the same argument for a given call to LALR_1.

This algorithm is straightforward and fairly efficient for computing the lookahead set for a single item in a given state. However, when it is used to compute the lookahead set for many final items in an LR(0) automaton, as is necessary for constructing an LALR(1) parser, much time is spent recomputing the same intermediate lookahead sets. One way to avoid these recomputations is to save all lookahead sets that are computed. However, this approach would not only require much space, but it cannot easily be incorporated in this framework because of the *global* nature of the algorithm. A more reasonable approach (space-wise) would be to save the intermediate $L(q, A)$ sets, but once again, it is not clear how the computation of these sets can be factored out in the above algorithm.

2.2.2 DeRemer and Penello

The LALR(1) lookahead algorithm of DeRemer and Penello (DP) is superior to the KM algorithm in that it avoids duplicating the computation of certain intermediate sets, called FOLLOW, which are analogous to the L sets of theorem 2.1.2. Additional space is required to save these FOLLOW sets, but the number of such sets needed is bounded by the number of nonterminal transitions in GOTO_0 which is usually not very large.

In 1977, Eve and Kurki-Suonio (EK) [14] presented an efficient algorithm for computing the transitive closure of an arbitrary relation based on Tarjan's algorithm [7] for finding strongly connected components in a directed graph. Recall that a strongly connected component(SCC) of a directed graph is a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex in the set. An SCC consisting of a single node with no path to itself is said to be *trivial.* The main contributions of DeRemer and Penello was in showing how the EK algorithm can be adapted to efficiently compute a recursively defined set-valued function, and, in particular, how to apply that algorithm to the computation of LALR(1) lookahead sets. The EK algorithm will be described later, in detail; but first, the fundamental DP definitions and theorems are reviewed.

Definition 2.2.2: (p, A) **reads** (r, C) iff $\text{GOTO}_0(p, A) = r$, $\text{GOTO}_0(r, C)$ is defined and $C \Rightarrow^* \varepsilon$.

The **reads** relation relates certain states of M_0 in the same way as states were related in definition 2.2.1 of TRANS sets. In fact, DP also define READ sets which are analogous to the TRANS sets in the following way:

$$\text{READ}(q, A) = \text{TRANS}(\text{GOTO}_0(q, A))$$

The following theorem which is similar to definition 2.2.1 is also presented:

Theorem 2.2.3:

$$\begin{aligned}\mathrm{READ}(p, A) &= \mathrm{DR}(p, A) \cup \bigcup\{\mathrm{READ}(r, C) \mid (p, A) \textbf{ reads } (r, C)\} \\ &\text{where} \\ \mathrm{DR}(p, A) &= \{a \in T \mid \mathrm{GOTO}_0(p, Aa) \text{ is defined}\}\end{aligned}$$

Definition 2.2.3: (p, A) **includes** (p', B) iff $B \to \beta A \gamma$, $\gamma \Rightarrow^* \varepsilon$, and $\mathrm{GOTO}_0(p', \beta) = p$.

The **includes** relation relates state-nonterminal pairs in the same way as they were related in theorem 2.2.2 by the L equation.

If a state p contains a transition on a nonterminal symbol A, then it also contains at least one item of the form $[A \to \cdot\omega]$ which was introduced by closure. The FOLLOW set of a state p and a nonterminal A is defined as follows:

$$\mathrm{FOLLOW}(p, A) = \mathrm{LALR}_1(p, [A \to \cdot\omega])$$

In other words, $\mathrm{FOLLOW}(p, A)$ is the set of all terminal symbols that may appear on input after a transition on A in state p. Similarly, the LALR(1) lookahead set for a state q and a final item $[A \to \omega\cdot]$ is captured by LA sets which are defined as follows:

$$\mathrm{LA}(q, [A \to \omega\cdot]) = \mathrm{LALR}_1(q, [A \to \omega\cdot])$$

The next two theorems of DP show how FOLLOW and LA sets can be computed:

Theorem 2.2.4:

$$\mathrm{FOLLOW}(p, A) = \mathrm{READ}(p, A) \cup \bigcup\{\mathrm{FOLLOW}(p', B) \mid (p, A) \textbf{ includes } (p', B)\}$$

Theorem 2.2.5:

$$\mathrm{LA}(q, A \to \omega\cdot) = \bigcup\{\mathrm{FOLLOW}(p, A) \mid p \in \mathrm{PRED}(q, \omega)\}$$

By combining theorems 2.2.5 and 2.2.4 and definition 2.2.3, one observes that the FOLLOW sets described above are the same as the L sets of theorem 2.2.2 and the LA sets of theorem 2.2.5 are also the same as the LALR_1 sets of therorem 2.2.2, except that the breakdown of the components is done differently.

This framework has many advantages. From a practical point of view, each intermediate FOLLOW sets can be computed once and saved for later use. As these intermediate sets are computed, their content can also reveal certain facts about the underlying grammar. DeRemer and Penello proved that when the **reads** relation contains one or more cycles, the underlying grammar is not LR(k), for any k. They also conjectured that given

(p, A), a nonterminal transition that is in a nontrivial SCC of the digraph induced by the **includes** relation, the corresponding grammar is not LR(k), for any k if $\mathrm{READ}(p, A) \neq \emptyset$. This conjecture was later proved by Sager [34].

From the above theorems, one observes that the problem of computing LALR(1) lookahead sets has been decomposed into four separate computations. In reverse order, they are as follows: LA sets are computed from FOLLOW sets of nonterminal transitions; FOLLOW sets are computed from READ sets of nonterminal transitions; READ sets are computed from DR (Direct Read) sets; and DR sets are computed by inspecting certain transitions of GOTO_0. In addition, two relations: **reads** and **includes** (defined on nonterminal transitions), relate the READ sets and the FOLLOW sets, respectively.

The way in which FOLLOW, READ and DR are related makes it possible for an appropriate graph traversal algorithm to be applied to compute READ from DR and FOLLOW from READ. The two graphs of interest are those induced by the relations **reads** and **includes**. The general case of this problem can be stated as follows.

Let R be a relation on a set X. Let F and F' be set-valued function such that for all $x \in X$,

$$F(x) = F'(x) \cup \bigcup\{F(y) \mid xRy\}$$

where F' is given for all $x \in X$.

If the underlying graph induced by the relation R contains no cycles, a straightforward recursive algorithm can efficiently compute F. If, however, the graph contains SCC's, a recursive algorithm such as the one advocated by Kristensen and Madsen can be relatively inefficient since it involves multiple traversal of the SCC's. If x and y are members of an SCC then xR^*y and yR^*x. It follows from that observation that $F(x) \subseteq F(y)$ and $F(y) \subseteq F(x)$; hence, $F(x) = F(y)$.

The algorithm proposed by DeRemer and Penello for computing F can be seen as accomplishing two tasks. The first task is to construct a new digraph by collapsing each set of nodes making up a non-trivial SCC into a single *supernode* and for each supernode set σ, let $F'(\sigma) = \bigcup\{F'(x) \mid x \in \sigma\}$. The new digraph so constructed, contains no cycles. The second task is to traverse the new digraph in a straightforward recursive fashion to compute F on the supernodes, then propagate the value of $F(\sigma)$ to each node x that was collapsed into σ. The striking efficiency of the algorithm is due to the fact that these two objectives are achieved in a single traversal of the original digraph without having to explicitly construct the collapsed graph.

Let S be an initially empty global stack of elements of X (the size of S will never exceed $|X|$). Let N be a global mapping from each element of X into a non-negative number. Let F and F' be global set-valued maps defined as above, where F' is precomputed or it can be easily computed *on the fly*. The DP digraph algorithm is stated in Figure 2.3.

TRAVERSE is a recursive procedure that takes as argument a node $x \in X$. Initially, the global map N is initialized to 0 for each element of x indicating that $F(x)$ has not yet been computed. When $1 \leq N(x) < \infty$, it indicates that the computation of $F(x)$ is in progress. When $N(x) = \infty$ it indicates that $F(x)$ has already been computed.

Upon entering TRAVERSE for a given node x, x is pushed into the global stack S, the number of elements in S is saved in an integer variable d, $N(x)$ is set to d and $F(x)$ is initialized with $F'(x)$. Next, the algorithm loops through the set of elements y related to

```
proc TRAVERSE(x)
    S := S + [x];
    d := #S;
    N(x) := d;
    F(x) := F'(x);
    for y $\in$ X | x R y loop
        if N(y) = $\Omega$ then
            TRAVERSE(y);
        end if;
        N(x) := N(x) min N(y);
        F(x) := F(x) $\cup$ F(y);
    end loop;
    if N(x) = d then
        until y = x loop
            pop y from S;
            F(y) := F(x);
            N(y) := $\infty$;
        end loop;
    end if;
end TRAVERSE;

N := $\emptyset$;
S := [ ];
for x $\in$ X | N(x) = $\Omega$ loop
    TRAVERSE(x, 1);
end loop;
```

Fig. 2.3: Digraph algorithm

x, and for each such y, if the computation of $F(y)$ had not yet been initiated, TRAVERSE is invoked recursively to compute it. If the computation of $F(y)$ is already in progress (as indicated by $N(y) < N(x)$), then the relation R contains a cycle which includes x and y. In that case $N(y)$ is assigned to $N(x)$ to indicate that the node y was traversed first and that the computation of $F(x)$ cannot be completed until $F(y)$ is completed. In any case, for each y such that xRy, the elements of $F(y)$, which must include (at least) the set $F'(y)$, are added to the set $F(x)$.

Upon exiting the loop, if $N(x) \neq d$ the procedure TRAVERSE exits, leaving x in the stack S. On the other hand, if $N(x) = d$, this indicates that x is the very first element of an SCC that was traversed and $N(x)$ has the lowest value of all $N(y)$ for some element y in the SCC. When TRAVERSE returns to the first element, x, of an SCC, all the elements of the SCC are on top of the stack S with x at the bottom of the pile and the computation of $F(x)$ is complete. The algorithm then proceeds with its final step by popping, in turn, each element y of the SCC from S, setting $F(y) = F(x)$ and setting $N(y) = \infty$.

To compute the READ sets using the digraph algorithm, let X be the set of state and nonterminal pairs (p, A) in the domain of GOTO_0; let F' be the function DR; and let R be the **reads** relation. To compute the FOLLOW sets, let X be the same set of pairs as for the READ sets; let F' be the READ map; and let R be the **includes** relation.

As can be observed from the two algorithms discussed so far, algorithms for computing LALR(1) lookahead sets are dominated by union operations. Using the number of union operations performed as the criterion for measuring the time efficiency of such algorithms, the digraph algorithm discussed in this section is faster than the KM algorithm. Given a digraph with n nodes and m edges, its worst-case running time is $\mathrm{O}(n + m)$. Using this method to compute all the LALR(1) lookahead sets for a given LR(0) automaton, the total number of union operations required is equal to the sum of the number of edges in the digraphs induced by the **reads** and **includes** relations for the automaton, plus the number of union operations required to compute the final LA sets. This approach also has the advantage of being incremental; i.e., a given FOLLOW set does not have to be computed unless it is required to compute a final lookahead set or it is related (via **includes**) to another FOLLOW set that is required. In practice, this feature is useful since not all lookahead sets need to be computed in order to resolve conflicts.

2.2.3 Park, Choe and Chang

Park, Choe and Chang sought to reduce the size of the graph induced by the **includes** relation as proposed by DeRemer and Penello by eliminating vertices and edges associated with closure items. The main idea behind their approach was to precompute the lookahead contribution of the FOLLOW sets associated with such items directly from the grammar using a *left dependency* relation $\mathbf{L} \subseteq N \times N$, defined as follows:

$$B \,\mathbf{L}\, C \quad \text{iff} \quad B \rightarrow C\beta \in P$$

The digraph induced by the L relation is called the L-graph. Each edge (B, C) of the L-graph is labeled with the suffix β.

Definition 2.2.4:

$$\mathrm{PATH}_k(B, C) = \bigcup\{\mathrm{FIRST}_k(\beta_n ... \beta_2\beta_1) \quad | \quad B_0 = B,\ B_n = C,\ n \geq 0,$$

$$B_i \to B_{i+1}\beta_{i+1} \in P,\ 0 \le i < n\}$$

where the sequence $\beta_1 ... \beta_n$ is the concatenation of the edge labels of the path $(B_0, ..., B_n)$ in the L-graph.

$\mathrm{PATH}_k(B, C)$ is a subset of the lookahead strings or initial prefixes thereof that will appear after a nonterminal C when an item $[A \to \alpha \cdot B\beta]$ is in a given state and $B \Rightarrow^* C\gamma$. Note that the definition of PATH_k is independent of the LR state containing these items.

Consider an operator δ, a mapping in the power set of LR(k) items, defined as follows:

Definition 2.2.5: Let $[A \to \alpha \cdot X\beta, u]$ be an LR(k) item; then,

$$\delta(\{[A \to \alpha \cdot X\beta, u]\}) = \{[X \to \cdot\omega, v] \mid v \in \mathrm{FIRST}_k(\beta u),\ X \to \omega \in P\}$$

The reflexive transitive closure of δ can be used to define the CLOSURE operation on an LR(k) item as follows:

$$\mathrm{CLOSURE}(\{[A \to \alpha \cdot X\beta, u]\}) = \delta^*(\{[A \to \alpha \cdot X\beta, u]\})$$

Clearly, these definitions can be extended to a set of LR(k) items containing more than one element. Therefore, given the kernel of a state p in a canonical collection of LR(k) items, the state itself can be characterized using δ, as follows:

$$p = \mathrm{KERNEL}(p)\ \cup\ \delta^+(\mathrm{KERNEL}(p))$$

where the two sets $\mathrm{KERNEL}(p)$ and $\delta^+(\mathrm{KERNEL}(p))$ are disjoint.

Given an LR(k) kernel item $[A \to \alpha \cdot B\beta, u]$, PCC proved the following lemma which relates the kernel item in question to all the closure items that it can introduce:

Lemma 2.2.4:

$$\delta^+(\{[A \to \alpha \cdot B\beta, u]\}) = \{[C \to \cdot\gamma, v] \mid B\ \mathbf{L}^*\ C,\\ v \in \mathrm{PATH}_k(B, C) \oplus_k \mathrm{FIRST}_k(\beta u),\ C \to \gamma \in P\}$$

From the above lemma, one observes that the set of LR(k) lookahead strings associated with each closure item can be expressed in terms of the PATH function and the suffix and lookahead set of the kernel item from which the closure item in question was derived. Thus, with this formalism, the computation of lookahead sets does not require steps involving intermediate closure items since their contribution is effectively captured by the PATH_k sets. Combining definition 2.1.5 with lemma 2.2.4, one can conclude that the same equation holds for the LALR(k) case, since PATH_k does not depend on states. The following theorem summarizes this fact:

Theorem 2.2.6: Let $p \in M_0$, $[A \to \alpha_1 \cdot \alpha_2] \in P$, and $A \neq S'$; then

$$\mathrm{LA}_k([A \to \alpha_1 \cdot \alpha_2], p) =\\ \{u \mid u \in \mathrm{PATH}_k(A', A) \oplus_k \mathrm{FIRST}_k(\beta_2) \oplus_k LA_k([B \to \beta_1 \cdot A'\beta_2], q),\\ q \in \mathrm{PRED}(p, \alpha_1),\ A'\ \mathbf{L}^*\ A, [B \to \beta_1 \cdot A'\beta_2] \in \mathrm{KERNEL}(q)\}$$

Even though their formalism is presented for the general case, Park, Choe and Chang only considered the case of $k = 1$ when describing their algorithm. Using theorem 2.2.6, they derived the following constructive definition of LALR(1) sets:

$$\begin{aligned}\mathrm{LA}([A \to \alpha_1 \cdot \alpha_2], p) = \{a \mid {} & a \in \mathrm{PATH}(A', A) \oplus \mathrm{FOLLOW}(q, A'),\\ & q \in \mathrm{PRED}(p, \alpha_1),\ A'\mathbf{L}^*A,\ [B \to \beta_1 \cdot A'\beta_2] \in \mathrm{KERNEL}(q)\}\end{aligned}$$

where

$$\begin{aligned}\mathrm{FOLLOW}(q, A') = \{a \mid {} & a \in \mathrm{FIRST}(\beta_2) \oplus \mathrm{PATH}(B', B) \oplus \mathrm{FOLLOW}(r, B'),\\ & [B \to \beta_1 \cdot A'\beta_2] \in \mathrm{KERNEL}(q),\ r \in \mathrm{PRED}(q, \beta_1),\\ & [C \to \gamma_1 \cdot B'\gamma_2] \in \mathrm{KERNEL}(r),\ B'\mathbf{L}^*B\}\end{aligned}$$

As was observed before, in the case $k = 1$, the $\oplus$ operation can be replaced by a conditional union operation. Once again, note that the only items involved in the above equations are kernel items.

The FOLLOW sets described above are not the same as those of DeRemer and Penello. Note, for example, that it may be the case that $\mathrm{READ}(q, A') \not\subseteq \mathrm{FOLLOW}(q, A')$ if state q also contains a closure item with A' as its dot symbol. But, in such a case, the contribution of these closure lookahead symbols is captured by $\mathrm{PATH}(A', A)$ in the LA equation. This distinction is not made clear in the PCC paper. In fact, the definition the authors give for FOLLOW (definition 5.1 in [31]) is exactly that of DeRemer and Penello, but in proving their result, they use a different definition though they refer to it as the same definition.

Note that the domain of the FOLLOW map described above is limited to pairs (q, A') where the state q in question contains at least one kernel item whose dot symbol is A'.

The PCC algorithm is also implemented in stages just like the DP algorithm. Firstly, the L-graph is constructed, then the reflexive transitive closure $\mathbf{L}^*$ of $\mathbf{L}$ is computed (using the digraph algorithm). Next, $\mathbf{L}^*$ is used to compute the necessary PATH sets. Then, the necessary FOLLOW sets are computed using the digraph algorithm. Finally, the PATH and FOLLOW sets are used to compute the LA sets.

Park, Choe and Chang claim that the efficiency of their approach is due to the "order of magnitude reduction" in the number of FOLLOW sets that are computed using their formalism. They also claim that though this saving is partially offset by the calculation of the PATH sets, in general, far fewer of these sets are required than FOLLOW sets since the former depend only on nonterminals, in contrast to the latter, involving states.

This author was not able to substantiate these claims. In fact, experiments with this method were consistently outperformed in time and storage utilization by the DP algorithm. In addition, the PCC approach has the major disadvantage that it cannot be implemented incrementally in a way that distributes the cost of computing each lookahead set somewhat uniformly. In other words, the most costly part of the computation is factored out by the construction of the L-graph and the calculation of the PATH sets which must be done globally. It is almost never the case that lookahead sets must be computed for all final items in order to resolve conflicts. Therefore, these global calculations are a waste of time and storage since they otherwise serve no other useful purpose (such as helping in the identification of certain non-LR(k) grammars). Furthermore, the algorithm also requires the computation of FIRST for certain suffixes. PCC do not make clear how these sets should be computed and discount their impact on the overall cost of the algorithm by making the erroneous statement that the "computation of FIRST is also required in constructing LR(0) states".

2.2.4 Bermudez and Logothetis

The three algorithms discussed, so far, have had the same general flavor: construct one or more grammar-based maps such as FIRST and PATH, construct the LR(0) machine, construct intermediate sets such as TRANS, READ and FOLLOW, and use them to compute lookahead sets for final items. The algorithm of Bermudez and Logothetis (BL) is much simpler. It is similar to the DP approach in that the same intermediate FOLLOW sets are computed, but their strategy is radically different. Before getting into the details of the BL algorithm, let's first review the definition of SLR parsers and how they are constructed.

In an SLR parser, the lookahead set for a final item is computed as the *traditional* FOLLOW set of terminal symbols that may follow the left side nonterminal in some sentential form. To avoid confusion, these sets will be referred to as SLR_FOLLOW sets.

Definition 2.2.6:

$$\text{SLR\_FOLLOW}(A) = \{a \mid S \Rightarrow^* \alpha A a \beta\}$$

Usually, SLR_FOLLOW sets are computed directly from the grammar using a simple iterative algorithm [32] [37], but they can also be computed using the digraph algorithm described earlier [25]. For a final item $[A \rightarrow \omega\cdot]$ in a state q, the SLR lookahead set is defined as:

$$\text{SLR\_LA}(q, [A \rightarrow \omega\cdot]) = \text{SLR\_FOLLOW}(A)$$

Note from the equation above that the computation of an SLR lookahead set does not depend on the state. The symbols in the LALR(1) lookahead set for $[A \rightarrow \omega\cdot]$ in state q also "follow" nonterminal A, but they do so in the context of state q as can be observed in theorem 2.2.5.

The technique of Bermudez and Logothetis consists of constructing a new grammar, G', which captures the "contextual dependency" of LALR(1) FOLLOW sets in such a way that the FOLLOW set for each state-nonterminal pair in the domain of GOTO_0^G corresponds to the SLR_FOLLOW set of a nonterminal in G'.

Recall the following property of LRM_0: for every state-nonterminal pair (p_1, A) in the domain of GOTO_0 and for every production $A \rightarrow \omega \in P$, there exists a state r such that $\text{GOTO}_0(p_1, \omega) = r$ and $[A \rightarrow \omega\cdot] \in r$.

The new grammar G' is constructed in such a way that its vocabulary, V, consists of all pairs in the domain of GOTO_0^G. For each each pair $[p_1, A]$ corresponding to a nonterminal transition and each rule $A \rightarrow \omega$ in the original grammar G, G' contains one production for each path corresponding to $\text{GOTO}_0^G(p_1, \omega)$. The left side of the production is the pair $[p_1, A]$. The right-hand side of the production consists of the set of pairs corresponding to the transitions on the symbols of ω; i.e., for each $\omega = X_1 X_2 ... X_n$, assume (WLOG) that $\text{GOTO}_0^G(p_i, X_i) = p_{i+1}$, $1 \leq i < n$; then G' contains the following production:

$$[p_1, A] \;\rightarrow\; [p_1, X_1][p_2, X_2]...[p_n, X_n]$$

In particular, if $\omega = \varepsilon$, then $[p_1, A] \rightarrow \varepsilon$ is also in G'. Thus, grammar G' is similar to G, except for the fact that a certain amount of symbol splitting has taken place during the construction of LRM_0. The definition of G' can be formalized as follows:

Definition 2.2.7: For a context-free grammar $G = (N, T, P, S)$, let $G' = (N', T', P', S')$, where

$$\begin{aligned}
N' &= \{[p, A] \mid \mathrm{GOTO}_0^G(p, A) \text{ is defined}\}\\
T' &= \{[p, a] \mid \mathrm{GOTO}_0^G(p, a) \text{ is defined}\}\\
S' &= [\mathrm{IS}_0, S]\\
P' &= \{[p_1, A] \rightarrow [p_1, X_1][p_2, X_2]...[p_n, X_n] \mid [p_1, A] \in N',\\
&\qquad [p_i, X_i] \in N' \cup T' \text{ for } 1 \le i \le n,\\
&\qquad \mathrm{GOTO}_0^G(p_i, X_i) = p_{i+1} \text{ for } 1 \le i < n\\
&\qquad \text{and } A \rightarrow X_1X_2...X_n \in P\}
\end{aligned}$$

In [32], the authors suggest that SLR_FOLLOW for all nonterminals A in a given grammar $G = (N, T, S, P)$ be computed by "applying the following rules until nothing can be added to any [SLR_]FOLLOW set":

1. Place $\perp$ in SLR_FOLLOW(S)
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in SLR_FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, then everything in SLR_FOLLOW(A) is in SLR_FOLLOW(B).

Once G' has been constructed and SLR_FOLLOW has been computed for the nonterminals of G', the LALR(1) lookahead set for a given state-item pair in LRM_0^G can be obtained as follows:

$$\mathrm{LA}(p, [A \rightarrow \omega\cdot]) = \{a \mid [r, a] \in \mathrm{SLR\_FOLLOW}([q, A]),\ q \in \mathrm{PRED}(p, \omega)\}$$

The BL algorithm is simple and straightforward. However, if implemented with the above iterative algorithm for SLR_FOLLOW, as suggested in [38], it will perform very poorly compared to the other algorithms. Fortunately, the digraph algorithm can also be used to compute the SLR_FOLLOW sets efficiently. This will be discussed in the next section.

2.3 Improvements

In this sections some modifications are suggested to improve the performance of the KM, DP and BL algorithms.

2.3.1 Improving the KM algorithm

As pointed out earlier, the global nature of the KM algorithm makes it difficult to make any fundamental change to that algorithm. However, some local optimizations are still possible that can greatly reduce the total number of union operations it requires. By

```
procedure TRANS(q);
    TM := TM ∪ {q}
    LA := LA ∪ {X | X ∈ $T$, [B → $\alpha$·X$\beta$] ∈ q}
    for [B → $\alpha$·X$\beta$] ∈ q loop
        if X $\Rightarrow^*$ $\varepsilon$ and (GOTO$_0$(q, X) ∉ TM) then
            TRANS(GOTO$_0$(q, X));
        end if;
    end loop;
end TRANS;

procedure LALR(p, [A → $\alpha$ · $\beta$]);
    for q ∈ PRED(p, $\alpha$) | [q, A] ∉ DONE loop
        DONE := DONE ∪ {[q, A]};
        TM := ∅;
        TRANS(GOTO$_0$(q, A));
        for [B → $\gamma$·A$\delta$] ∈ q | $\delta$ $\Rightarrow^*$ $\varepsilon$ loop
            LALR([B → $\gamma$·A$\delta$], q);
        end loop;
    end loop;
end LALR;
```

Fig. 2.4: Improved TRANS and LALR procedures

observing that $\text{LALR}_1(q, [A \rightarrow \cdot\alpha]) = L(q, A)$, (essentially the FOLLOW sets) Kristensen and Madsen reformulated the LALR procedure of Figure 2.2, using the set DONE to keep track of (state, nonterminal) pairs instead of (state, item) pairs. The time performance of the KM algorithm can be further improved by precomputing, for each state q, the set of terminals on which a transition can be made on q and adding these terminals to related LA sets with a single union operation. Figure 2.4 depicts new formulations for the LALR and TRANS procedures that incorporate these two changes. If the LA and TRANS sets are implemented as bit-strings, this new approach can significantly improve the running time of the KM algorithm since, in many cases, adding a single element to a bit set (as in Figure 2.2) can be as expensive as unioning two bit sets.

2.3.2 Improving the DP algorithm

Earlier, it was shown that the READ sets proposed by DP are related to the TRANS sets proposed by KM as follows:

$$\text{READ}(q, A) = \text{TRANS}(\text{GOTO}_0(q, A)).$$

From this equation, one observes that a TRANS set is defined on each state whose incoming edges are labeled by a nonterminal, where as a READ set is defined on each each edge pointing to such a state. Therefore, READ sets can be computed more efficiently by first computing the TRANS sets for the relevant states and propagating these sets along the incoming edges for the READ sets.

The digraph algorithm can be used instead of the recursive KM algorithm to compute the TRANS sets. To do this, the **reads** relation and the DR function must be defined on states rather than state-nonterminal pairs. The new definitions follow:

Definition 2.3.1: q **reads** r iff $\text{GOTO}_0(q, A) = r$ and $A \Rightarrow^* \varepsilon$.

Definition 2.3.2: $\text{DR}(q) = \{a \in T \mid [B \rightarrow \beta \cdot a\gamma] \in p\}$

Lemma 2.3.1: $\text{TRANS}(q) = \text{DR}(q) \cup \bigcup\{\text{TRANS}(r) \mid p \textbf{ reads } r\}$

Using Lemma 2.3.1, the instantiation of the digraph algorithm for TRANS is straightforward. Let X be the set of states $p \in M_0$ such that the incoming symbol on which a transition is made into p is a nonterminal. Let R and F' be the **reads** relation and DR function of definitions 2.3.1 and 2.3.2, respectively.

However, a better overall approach is to compute TRANS sets from FIRST sets as indicated in lemma 2.2.3. The relevant FIRST sets for this purpose are the sets FIRST(β) for each suffix β that appears after a nonterminal in a production of G. This approach avoids the construction of the digraph induced by the **reads** relation. Moreover, given the relevant FIRST sets, the FOLLOW sets can also be computed without the explicit construction of the digraph induced by the **includes** relation (as proposed in [26]), since that relation can be computed on the fly in such a case. Recall from definition 2.2.3 that the inclusion of two pairs (p, A) and (p', B) in the **includes** relation is based on a *nullability* test of a suffix following a nonterminal. This is equivalent to testing for the presence of ϵ in the FIRST set of the relevant suffix. This approach for computing READ and FOLLOW sets was used succesfully in [25]. It usually requires fewer union operations that the standard DP approach and the space overhead is much lower.

2.3.3 Computation of FIRST sets

For a grammar $G = (T, N, P, S)$, Knuth [5] defined the following *left-dependency* relation on grammar symbols:

$$X\ l\ Y \text{ iff } X \rightarrow X_1 X_2 ... X_n Y\alpha \text{ and } X_i \Rightarrow^+ \varepsilon, 1 \leq i \leq n$$

and showed that for each nonterminal A,

$$\text{FIRST}(A) = \{a \in T \mid A\ l^+\ a\}.$$

It is this problem that motivated the transitive closure algorithm of Eve and Kurki-Suonio [14], from which the DP digraph algorithm is derived. These autors proposed that FIRST_1 be computed for nonterminals in two steps. In step one, the transitive closure of the left-dependency relation is computed and for each SCC all its nodes are mapped onto a single representative node to obtain a directed graph that is free of cycles. "The resulting graph is then explored by Knuth's efficient "topological sort" algorithm" to obtain the final result.

In fact, the digraph algorithm, as described earlier, can be used to compute FIRST_1 for nonterminals in a single pass without having to compute a transitive closure first. The set on the right side of the equation above can be viewed as the union of two sets: an initial set consisting of terminal symbols a that left-depend directly on A and a set of terminal symbols that are contributed by other nonterminals that left-depend directly on A. Hence, the equation can be rewritten as:

$$\mathrm{FIRST}(A) = \{a \in T \mid A \; l \; a\} \cup \bigcup\{\mathrm{FIRST}(B) \mid A \; l \; B\}.$$

This new formulation of $\mathrm{FIRST}_1(A)$ is exactly in a form suitable for the digraph algorithm. Once, FIRST_1 is computed for each nonterminal in a grammar, it can be extended for arbitrary strings [32].

2.3.4 Improving the BL algorithm

The computation of the SLR_FOLLOW sets is central to the BL algorithm. Thus, any improvement in the computation of these sets will result in an overall improvement of the BL algorithm. From rule 3 of the iterative SLR_FOLLOW algorithm proposed earlier, one observes that certain nonterminals are related to others and that relation determines how the corresponding SLR_FOLLOW sets of these nonterminals are computed. The relation in question, called **s_includes** here, can be defined as follows:

Definition 2.3.3: B **s_includes** A iff $A \rightarrow \alpha B \beta \in P$ and $\beta \Rightarrow^* \varepsilon$.

For each nonterminal B, define a function DF (read Directly Follows) on B as follows:

$$\mathrm{DF}(B) \quad = \quad \bigcup\{\mathrm{FIRST}(\beta) \mid A \rightarrow \alpha B \beta \in P\} - \{\varepsilon\} \tag{2.5}$$

$\mathrm{DF}(B)$ is simply the subset of SLR_FOLLOW(B) obtained from applying rule 2 of the iterative algorithm. Given the **s_includes** relation and the DF function, the SLR_FOLLOW sets can be written in equation form as follows:

$$\mathrm{SLR\_FOLLOW}(B) \quad = \quad \{\perp\}, \text{ if } B = S'. \text{ Otherwise,} \tag{2.6}$$

$$\mathrm{SLR\_FOLLOW}(B) \quad = \quad \mathrm{DF}(B) \cup \bigcup\{\mathrm{SLR\_FOLLOW}(A) \mid B \;\mathbf{s\_includes}\; A\}. \tag{2.7}$$

With the above formulation, the digraph algorithm can be instantiated in the usual fashion to compute the SLR_FOLLOW sets. When this approach is used, the computation of lookahead sets with the BL method requires the same number of union operations as the improved DP algorithm described in section 2.3.2.

Let G be a context-free grammar and consider the BL grammar $G' = (N', T', P', S')$ obtained from G. It is not hard to see that a nonterminal $[p, B] \in N'$ **s_includes** $[q, A]$ if and only if the corresponding state-nonterminal pair (p, B) in the domain of GOTO_0^G **includes** (q, A). Since there is a one-to-one and onto correspondence between each nonterminal $[p, B] \in N'$ and a pair (p, B) in the domain of GOTO_0^G, the graph induced by the **includes** relation is isomorphic to the graph induced by the **s_includes** relation. Therefore, given the READ sets for all nonterminal transitions in GOTO_0^G and the DF sets for all nonterminals in N', the same number of union operations required to compute

SLR_FOLLOW on nonterminals of G' is required to compute FOLLOW sets for all (state, nonterminal) pairs in the domain of GOTO_0^G.

For each state q in M_0^G whose incoming edges are labeled by a nonterminal B, the kernel set of q contains a set of items of the form $[A \to \alpha B \cdot \beta]$. As suggested in section 2.3.2, the TRANS set of q (from which relevant READ sets are obtained) is computed as the union of FIRST(β) for all suffix β following B in the kernel items of q. Similarly, assume that $\mathrm{GOTO}_0^G(p, B) = q$, the computation of DF($[p : B]$) (equation 2.5) is obtained by forming the union of FIRST(γ) for all suffix γ following $[p : B]$ in P'. Once again, since there is a one-to-one and onto correspondence between each rule in G' with $[p : B]$ in its right-hand side and a pair (p, B) such that $\mathrm{GOTO}_0^G(p, B) = q$, the number of union operations required to compute the TRANS sets is the same as is required to compute the DF sets.

Thus, when the digraph algorithm is used to compute SLR_FOLLOW sets, the BL algorithm requires exactly the same number of union operations as the improved DP algorithm. However, since extra time as well as space overhead is incurred in constructing G', the DP algorithm gives a better overall performance. Depending on the nature of the grammar, the extra space needed for G' may be substantial.

2.4 Remarks

In this section, four algorithms for computing LALR(1) lookahead sets have been reviewed. The first, by Kristensen and Madsen(KM) [24], is a straightforward recursive algorithm which is efficient for computing an individual lookahead set. However, in constructing an LALR(1) parser, a large number of lookahead sets must be computed and the computation of some of these sets usually depend on others. In such a situation, the KM algorithm performs poorly because it does not avoid recomputing any lookahead set. The second algorithm, by DeRemer and Penello(DP) [26], is the most efficient of all the algorithms mentioned. It is superior to the KM algorithm in that it avoids recomputing certain intermediate lookahead sets called FOLLOW, though it requires (a reasonable amount of) extra space in order to do so. The DP framework also allows one to detect, in certain cases, whether or not the underlying grammar is not $LR(k)$ for any k. (In general, this problem is undecidable.) The third algorithm, by Park, Choe and Chang(PCC) [31], is less time- and space-efficient than the DP algorithm, notwithstanding some experimental comparisons presented by these authors which would indicate otherwise. The last algorithm, by Bermudez and Logothetis(BL) [38], is the simplest of all the algorithms. A new grammar G' is derived from the LRM_0^G machine for the original grammar G, in such a way that the same intermediate FOLLOW sets advocated by DP can be computed more easily from G'. When implemented as suggested in the previous section, the BL and DP algorithms have essentially the same running time performance except for the overhead incurred in constructing the G' grammar. Experimental results obtained from using each of these algorithms are presented in Appendix A.

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.

```
bnf    →  rlist
rlist  →  ϵ
       |  rlist rule
rule   →  s '→' slist
slist  →  ϵ
       |  slist s
```

(a)

```
bnf    →  ϵ
       |  rlist
rlist  →  head
       |  rlist s
       |  rlist head
head   →  s '→'
```

(b)

```
bnf    →  rlist
rlist  →  ϵ
       |  rlist ; rule
rule   →  s '→' slist
slist  →  ϵ
       |  slist s
```

(c)

Fig. 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:

```
then  →  ϵ
      |  THEN
```

Similarly, observe that the grammar of Figure 3.1(a) can be modified into the LALR(1)

	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						

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

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

	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					

Fig. 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				

Fig. 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						

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

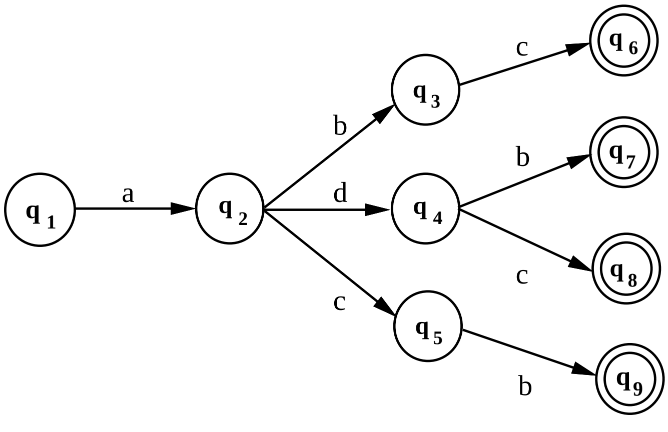


Fig. 3.6: DFA for {abc, adb, adc, acb}

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 [32].

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.

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

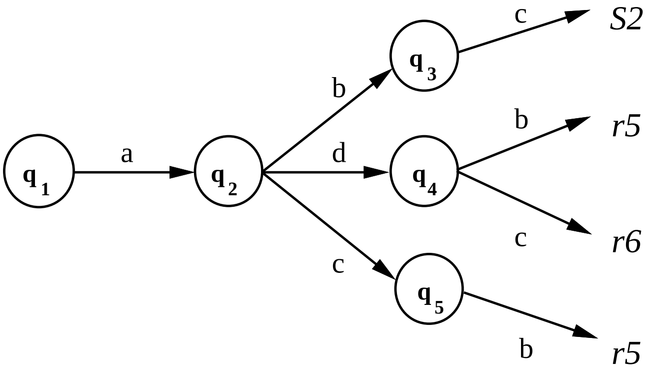


Fig. 3.7: Lookahead DFA

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 *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

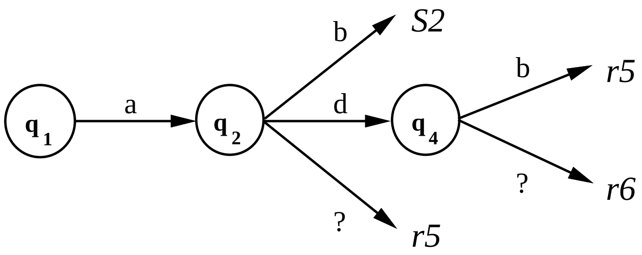


Fig. 3.8: Lookahead DFA with default

		ACTION				GOTO			
		?	→	s	⊥	bnf	rlist	rule	slist
ACTION$_1$:	1	R2				3	2		
	2	R1		S4				5	
	3				acc				
	4		S6						
	5	R3							
	6	R5							7
	7	R4		L9					
	8	R6							
Lookahead:	9	R4		S8	S8				

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

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, 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, $\text{READ}_k(p, X)$ is the set of all strings of length k that can be read following a transition on X in p; $\text{SHORT}_k(p, X)$ is the set of pairs $[w, [B \rightarrow \alpha \cdot X\beta]]$ where $[B \rightarrow \alpha \cdot X\beta]$ is an item in the state p and w is a non-empty string of length less than k that is derivable from β; $\text{READ}_{k^*}(p, X)$ is simply the union of the sets $\text{FIRST}_k(\beta)$ for all items $[B \rightarrow \alpha \cdot X\beta]$ in p less the empty string: ε. The set of strings in $\text{READ}_{k^*}(p, X)$ 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:
$\text{FOLLOW}_k(p, A) = \text{READ}_k(p, A)$
$\cup \bigcup\{\text{FOLLOW}_k(p', B) \mid (p, A)$ **includes** $(p', B)\}$
$\cup \bigcup\{\{w\}.\text{FOLLOW}_{k-|w|}(p', B) \mid [w, [B \rightarrow \alpha \cdot A\beta]] \in \text{SHORT}_k(p, A),\ p' \in \text{PRED}(p, \alpha)\}.$

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

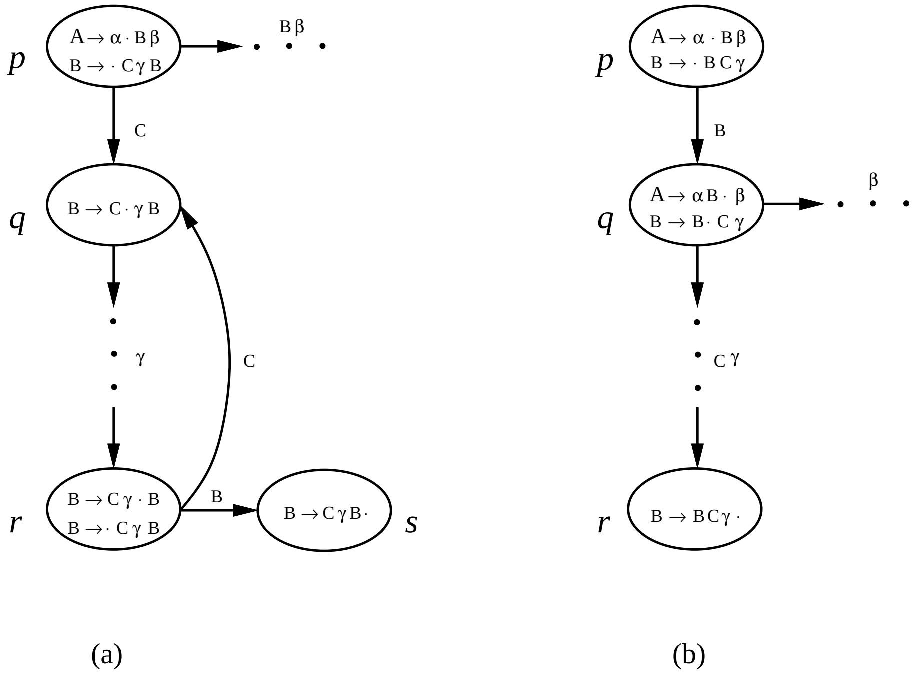
Let *stack* be a non-empty sequence of states representing a path in the LR(0) automaton of a grammar G and let X be a grammar symbol in G, $X \neq \epsilon$ and $X \neq S'$. A string w is said to be readable in the context of $(stack, X)$, if Xw can be successfully parsed starting from the state p on 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.



(a) (b)

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

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 $(stack, X)$:

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

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

```
function READ$_*$(stack, X, k);
    if k = 0 then
        return $\{\varepsilon\}$;
    end if;
    rd := $\emptyset$;
    ts := stack(#stack);
    q := GOTO$_0$(ts, X);
    for a $\in$ DR(ts, X) loop
        rd := rd $\cup$ {a.x | x $\in$ READ$_*$(stack+[q], a, k-1)};
    end loop;
    for (q, Y) | (ts, X) reads (q, Y) loop
        rd := rd $\cup$ READ$_*$(stack + [q], Y, k);
    end loop;
    for [C $\rightarrow \gamma\cdot$ X] $\in$ ts | C $\neq$ S' and $|\gamma| + 1 <$ #stack loop
        rd := rd $\cup$ READ$_*$(stack(1..#stack - $|\gamma|$), C, k)}
    end loop;
    return rd;
end READ$_*$;
```

Fig. 3.11: Recursive READ_{k^*} function

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 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

```
function READ_STEP(stack, X);
    configs := $\emptyset$;
    ts := stack(#stack);
    q := GOTO$_0$(ts, X);
    for a $\in$ DR(ts, X) loop
        configs := configs $\cup$ {[stack+[q], a]};
    end loop;
    for (q, Y) | (ts, X) reads (q, Y) loop
        configs := configs $\cup$ READ_STEP(stack+[q], Y);
    end loop;
    for [C $\rightarrow$ $\gamma$·X] $\in$ ts | C $\neq$ S' and |$\gamma$| + 1 < #stack loop
        configs := configs $\cup$ READ_STEP(stack(1..#stack - |$\gamma$|), C);
    end loop;
    return configs;
end READ_STEP;

function READ$_*$(stack, X, k);
    if k = 0 then
        return {$\varepsilon$};
    end if;
    rd := $\emptyset$;
    for [stk, a] $\in$ READ_STEP(stack, X) loop
        rd := rd $\cup$ {a.x | x $\in$ READ$_*$(stk, a, k-1)}
    end loop;
    return rd;
end READ$_*$;
```

Fig. 3.12: Incremental READ$_{k^*}$ function

suffix βY generates at least one string whose length is shorter than k. (Note that βY may 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 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 FOLLOW$_k(p, A)$ is based on the sets READ$_k(p, A)$ and SHORT$_k(p, A)$. Each element $w \in$ READ$_k(p, A)$, is added directly to FOLLOW$_k(p, A)$ and each element of 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 $w \in$ 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

```
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 := GOTO0(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;
```

Fig. 3.13: FOLLOW$_k$ *function*

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:
$\mathrm{FOLLOW}_0(stack, X) = \{\varepsilon\}$
$\mathrm{FOLLOW}_k(stack, X) =$

$$\bigcup\{\{a\}.\mathrm{FOLLOW}_{(k-1)}(stack + [q], a) \mid a \in \mathrm{DR}(ts, X),\ q = \mathrm{GOTO}_0(ts, X)\}$$
$$\cup \bigcup\{\mathrm{FOLLOW}_k(stack + [q], Y) \mid (ts, X)\ \mathbf{reads}\ (q, Y)\}$$
$$\cup \bigcup\{\mathrm{FOLLOW}_k(stack(1..\#stack - |\gamma|), C) \mid [C \to \gamma \cdot X] \in ts,\ |\gamma| + 1 < \#stack\}$$
$$\cup \bigcup\{\mathrm{FOLLOW}_k([q], C) \mid [C \to \gamma_1\gamma_2 \cdot X] \in ts,\ |\gamma_2| + 1 = \#stack,\ q \in \mathrm{PRED}(stack(1), \gamma_1)\}.$$

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 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 $\mathrm{GOTO}_0(q, \epsilon) = q$, 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 $[A \to \omega\cdot]$ in a state q is the following set:

```
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 → $\gamma$·X] ∈ ts | C ≠ S' loop
        if $|\gamma| + 1 <$ #stack then
            stacks := stacks ∪ FOLLOW_SOURCES(stack(1..#stack - $|\gamma|$), C, a)}
        else
            assert $\gamma = \gamma_1\gamma_2$ where $|\gamma_2| + 1 =$ #stack;
            for q ∈ PRED(stack(1), $\gamma_1$) loop
                stacks := stacks ∪ FOLLOW_SOURCES([q], C, a);
            end loop;
        end if;
    end loop;
    return stacks;
end FOLLOW_SOURCES;
```

Fig. 3.14: FOLLOW_SOURCES *function*

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

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

```
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;
```

Fig. 3.15: NEXT_LA *function*

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, $\#\text{ACTION}(q, a) <= 1$, $\forall a \in T$. 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.

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 $\text{ACTION}(q, t)$ 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 $\text{ACTION}(p, a) = \emptyset$, $\forall a \in T$ and resets $\text{ACTION}(q, t)$ 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, $\text{ACTION}(p, a)$ is updated accordingly. If no conflicts are detected in the new lookahead sets, the original

```
procedure LOOK_AHEAD(q)
    for $a \in T$ loop
        ACTION(q, a) := if (p := GOTO$_0$(q,a)) $\neq \Omega$ then {shift p} else $\emptyset$ end;
    end loop;
    for each final item A $\rightarrow \omega\cdot \in$ q loop
        for a $\in$ LA(q, A $\rightarrow \omega\cdot$) loop
            ACTION(q, a) := ACTION(q, a) $\cup$ {reduce A $\rightarrow \omega$};
        end loop;
    end loop;
    for a $\in T$ | #ACTION(q, a) > 1 loop
        sources := $\emptyset$;
        for act $\in$ ACTION(q, a) loop
            if act is a shift action then
                sources(act) := {[q]};
            else
                assert act = reduce A $\rightarrow \omega$;
                sources(act) := $\emptyset$;
                for p $\in$ PRED(q, $\omega$) loop
                    VISITED := $\emptyset$;
                    sources(act) := sources(act) $\cup$ FOLLOW_SOURCES([p],A,a);
                end loop;
            end if;
        end loop;
        RESOLVE_CONFLICTS(q, a, sources, 2);
    end loop;
    return;
end LOOK_AHEAD;
```

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

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 inconsistent 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.

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

Fig. 3.17: RESOLVE_CONFLICTS procedure

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:

```
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. Unfortunately, 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.

4. LALR(K) PARSER AND ERROR RECOVERY

4.1 Overview

4.1.1 The Parsing Framework

An LR *parsing configuration* has two components: a state stack and the remaining input tokens. This method assumes a framework in which the parser maintains a state stack, denoted state_stack, and an input buffer containing a fixed number of input symbols. These symbols include the current token or *lookahead*, denoted *curtok* and the token immediately preceding the current token (last token processed), denoted *prevtok*. The remaining tokens in the buffer are input tokens following *curtok*. A number of attributes are associated with each input symbol such as its class, its location within the input source, its character string representation, etc. An input symbol together with all its attributes is referred to as a *token element.* Each state q in the state stack is also associated with certain attributes including the grammar symbol that caused the transition into q (called the *in_symbol* of q), and the location of the first input token on which an action was executed in q.

An LR parsing configuration may be represented by a string of the form:

$$q_1, q_2, \ldots, q_m \quad | \quad t_1, t_2, \ldots, t_n.$$

The sequence to the left of the vertical bar is the content of the state stack, with q_m at the top; $q_1 \ldots q_m$ is a valid sequence of states in the LR parsing machine corresponding to a viable prefix. The sequence to the right of the vertical bar is the unexpended input. Each element t_i represents the class of a corresponding input symbol. The symbol t_1 represents the class of the current token, t_2 represents the class of the successor of the current token, etc. The symbol t_0 which is not shown above represents the class of previous token.

For simplicity, it will be assumed that the grammar used to construct the parser is LR(1), but this method is applicable to all forms of LR(k) parsers.

4.1.2 Error Recovery

A parsing configuration in which no legal action is possible is called an *error configuration.* When an error configuration is reached, the error recovery procedure is invoked. Its role is to adjust the configuration so as to allow the parser to advance a minimum predetermined distance in the input stream, usually two or three tokens past the repair point. The token on which the error is detected is referred to as the *error token* and the state in which the error is detected is called the *error state.*

Three kinds of recovery strategies are used. They are:

- *Simple recovery.* A simple recovery (also called *first level recovery* [21]) is a single symbol modification of the source text; i.e., the insertion of a single symbol into

```
     1. program TEST(INPUT, OUTPUT);
     2.     var X,Y: array[] of integer;
                              ^
***Error: index_type_list expected after this token
     3. begin
     4. 1:   x := y,
                   ^
***Error: ; expected instead of this token
     5.      if x == b then begin
                   ^
***Error: Unexpected symbol ignored
     6.         go to 1;
                <--->
***Error: Symbols merged to form GOTO
     7.      a := ((b + c)
                   ^-----^
                  ^------^
***Error: ")" inserted to complete phrase
***Error: "END" inserted to complete phrase started at line 5, column 21

     8. end.
```

Fig. 4.1: Primary phase recoveries

the input stream, the deletion of an input token, the substitution of a grammar symbol for an input token or the merging of two adjacent tokens to form a single one. Previous authors [21][36] have used a more restricted form of simple recovery involving only terminal symbols as repair candidates.

- *Phrase-level recovery.* In phrase-level recovery, the error procedure tries to recover by either deleting as small a sequence of tokens as possible in the vicinity of the error token or replacing such a sequence with a nonterminal symbol. This technique can be viewed as an *automatic* generalization of the *error productions* method described in [9].

- *Scope recovery.* A scope is a syntactically nested structure such as a parenthesized expression, a block or a procedure. In scope recovery, the strategy is to recover by inserting relevant symbols into the text to complete the construction of incomplete scopes.

This error recovery scheme consists of two phases called *Primary phase* and *Secondary phase.* In the *Primary phase*, an attempt is made to recover with minimal modification of the remaining input stream. Repairs that are attempted in the primary phase include the different kinds of simple recoveries, scope recoveries that require the deletion of no or one input symbol and phrase-level recoveries that do not require any deletion of input symbols. Figure 4.1 shows some examples of primary phase recoveries. In the *Secondary phase*, more radical approaches involving removal of some left context (state stack) information as well

```
     1. program P(INPUT,OUTPUT);
     2.     procedure FACTORIAL(X:INTEGER):integer;
                                          <------>
***Error: Unexpected input discarded
     3.     begin
     4.     end;
     5. begin
     6.    if count[listdata[sub] := 0 then
                   ^-----------^
                                  ^

***Error: "]" inserted to complete phrase
***Error: invalid relational_operator
     7.        a := ((b + c ]];
                     ^----^
                    ^-----^

                            <>
***Error: ")" inserted to complete phrase
***Error: ")" inserted to complete phrase
***Error: Unexpected input discarded
     8. end.
```

Fig. 4.2: Secondary phase recoveries

as multiple deletion of tokens from the remaining input (right context) are attempted. Figure 4.2 shows some examples of secondary phase recoveries.

4.1.3 Error Detection

A canonical LR(k) parser has the capability of detecting an error at the earliest possible point. However, as mentioned earlier, canonical LR(k) parsers are seldom used because of their size. Instead, variants such as LALR(k) and SLR(k) (usually $k = 1$) are used. These LR variants, in part, solve the space problem by always using the underlying LR(0) automaton. However, certain states in these parsers usually contain reduce actions that may be illegal, depending on the actual context. Illegal reduce actions do not cause the resulting parser to accept illegal inputs, but they prevent it from always detecting errors at the earliest possible point. This problem is usually compounded by the use of the space-saving technique known as *default reduction* described in the previous chapter. Another undesirable side effect of default reductions is that when they are used, it is no longer possible to compute, from the parsing table, the set of terminal symbols on which valid actions are defined in a given state. The inability to detect errors as soon as possible and to obtain a set of viable terminal candidates for a given state is very problematic for error recovery.

Furthermore, even with a canonical LR(k) parser, the ability to detect an error at the *earliest possible point* only guarantees that the prefix parsed up to that point is correct. Therefore, it is possible that the token on which an error is detected is not the one that is actually in error. Consider the following Pascal declaration:

```
FUNCTION F(X:TINY, Y:BIG, Z:REAL);
```

In this example, it is very difficult to deduce the actual intention of the programmer, but a simple substitution of the keyword "PROCEDURE" for the keyword "FUNCTION" would solve the problem. However, the error is not detected until the semicolon (;) is encountered or 15 tokens later.

In [36], Burke and Fisher introduced a *deferred parsing* technique where two parsers are run concurrently: one that parses normally and another that is kept at a fixed distance (measured in terminal symbols) back. When an error is encountered, error recovery is attempted at all points between the two parsers. This approach avoids the premature reductions problem and solves, in part, the problem of late detection of errors. However, the overhead of the two parsers penalizes correct programs.

In this method, a new LR driver routine called *deferred driver* is introduced. This new driver can effectively detect an error at the earliest possible point even if the parser contains default reductions. It can also be adapted to defer parsing actions on a fixed number of tokens with negligible slow-down on correct programs. To achieve this goal, an additional state stack is required for each deferred symbol. Thus, in practice, one must restrict the number of symbols on which actions are deferred.

The method also relies on having two mappings: *t_symbols* and *nt_symbols*, statically constructed, which yield for each state, a subset of the terminal and nonterminal symbols, respectively, on which an action is defined in the state in question. These subsets are the smallest subsets of viable error recovery *candidates* for each state. Their computation and optimization will be discussed in chapter 5.

The remainder of this chapter is organized as follows:

- detailed description of the new driver
- presentation of various recovery techniques
- discussion of how to apply these recovery techniques and how to issue accurate diagnostics.

4.2 The Driver

An important improvement that can be made to an LR(k) automaton is the removal of *LR(0) reduce states*. An LR(0) reduce state is a state that consists of a single final item. Therefore, such a state contains only reduce actions by the rule from which the final item in question is derived. If a representation of the parsing tables with default action is used, the parser will never consult the lookahead symbol when it is in one of these states. Thus, such states may be completely removed from the parser by introducing a new parsing action: *read-reduce*, which comprises a read transition followed by a reduction. If q is an LR(0) reduce state consisting of a single final item $[A \rightarrow \alpha X \cdot]$, in order to remove q from the automaton, the parsing tables can be modified as follows. For all states p such that $\mathrm{GOTO}_0(p, X) = q$, change the action in p on X to a read-reduce action by the rule $A \rightarrow \alpha X$. State q is now inaccessible and can be discarded. A read-reduce action is referred to as a *shift-reduce* when the symbol X in question is a terminal symbol and as a *goto-reduce* action when X is a nonterminal.

When the parser encounters a shift-reduce action by a rule $A \rightarrow \alpha t_1$ in a configuration $q_1, q_2, \ldots, q_m \mid t_1, t_2, \ldots, t_n$, it pops $|\alpha|$ elements from the state sequence, reads the input symbol t_1 and executes the appropriate nonterminal action on the state $q_{m-|\alpha|}$ and A. A goto-reduce action is processed in a similar fashion except that no input symbol is read in. From now on, it is assumed that all LR(0) reduce states have been removed from a given parsing table and replaced with read-reduce actions.

As there is only one kind of action that can be executed in an LR(0) reduce state, the removal of these states from an LR automaton does not cause *premature* reductions. Moreover, observe that when the parser computes a read-reduce action, that action is followed by a sequence of zero or more goto-reduce actions, and finally, by a goto action. All these actions may also be executed without deferral, by the same argument.

When the parser executes a reduce action in a non-LR(0) reduce state, that action is also followed by goto-reduce actions and a final goto action. If the reduce action in question is an illegal action, executed by default, then all the associated goto-reduce and goto actions following it are also illegal moves. To complicate matters, the goto action may be followed by a sequence of reduce actions on empty rules, each followed by its associated goto-reduces and a final goto action. In such a case, all actions induced by the lookahead symbol must be invalidated and the original configuration of the parser (prior to the initial reduction) must be restored.

One way to achieve this goal is as follows. When a reduce action is encountered, copy the state stack into a temporary stack and simulate the parser using the temporary stack until either a shift, shift-reduce or error action is computed on the lookahead symbol. If the first non-reduce action computed on the lookahead is valid, the temporary stack is copied into the state stack and the parsing can continue. Otherwise, the error recovery routine is invoked with the unadulterated state stack. This idea captures the essence of what needs to be done, but it is too costly for practical use.

Instead of copying the information, the temporary stack is used to hold the values of the contiguous elements of the state stack that have been added or rewritten. If the moves turn out to be valid, then only the added or rewritten elements are copied to the state stack. Otherwise, the original configuration is passed to the error recovery routine. This idea is illustrated in the *lookahead_action* function of Figure 4.3.

Starting with a given configuration, the *lookahead_action* function, in essence, checks whether or not it is possible to advance the parse past the current token, and if so, it returns the first non-reduce action induced by the current token. As a side effect, it also computes the index position (pos) of the topmost element in the state stack sequence that is still useful after all the actions induced by the current token are executed. All new stack information above that index position is stored in the temporary stack in its relative position.

Lemma 4.2.1: Let stack_top=top+1 prior to entering the inner loop of *lookaheadaction*, when the inner loop is exited, top $\leq$ stack_top.

Proof: The first iteration of the loop processes an initial reduce action. At the end of the first iteration, the variable top is either increased by 1 (in the case of an empty production, RHS(act) = 0), left unchanged (in the case of a unit production), or decreased. Each

```
-- Assume RHS and LHS are maps that yield the size of the right-hand side and left-hand side
-- symbol of a given rule, respectively. ACTION and GOTO are the terminal and nonterminal
-- parsing functions, respectively.

function lookahead_action(stack, tok, pos);
    temp_stack := [ ];
    pos := #stack;
    top := pos - 1;
    act := ACTION(stack(pos), tok);
    while act is a reduce action loop
        until act is not a goto-reduce action loop
            top := top - RHS(act) + 1;
            if top > pos then
                s := temp_stack(top);
            else s := stack(top);
            end if;
            act := GOTO(s, LHS(act));
        end loop;
        temp_stack(top+1) := act;
        act := ACTION(act, tok);
        pos := min(pos, top);
    end loop
    return act;
end lookahead_action;
```

Fig. 4.3: lookahead_action function

```
act := $q_0$;
state_stack := [ ];
location_stack := [ ];
while act is not the accept action loop
    state_stack := state_stack + [act];
    location_stack(#state_stack) := curtok.location;
    act := lookahead_action(stack, $t_1$, pos);        - - recall that $t_1$ = curtok.class
    if act $\neq$ error action then
        state_stack(pos+1..) := temp_stack(pos+1..);
        location_stack(pos+1..) := [curtok.location : i in [pos+1..#state.stack]];
        if act is a shift-reduce action then
            top := #state_stack;
            while act is a goto-reduce action loop
                top := top - RHS(act) + 1;
                act := GOTO(state_stack(top), LHS(act));
            end loop;
            stack(top+1..) := [ ];
        end if;
        get next token;
    else
        error_recovery();
    end if;
end loop;
```

Fig. 4.4: Driver with one deferred token

subsequent iteration of the loop is induced by a goto-reduce action processing a non-empty rule which either leaves the value of top unchanged or decreases it.

When the inner loop is exited, the transition state associated with the goto action that caused the exit is stored in the temporary stack at position top+1, the new upper bound of the stack. If top is a new upper bound for the initial state stack, pos is updated accordingly. When the outer loop is exited, the action that caused the exit is returned.

Assume that the starting state of an LALR(k) parser is denoted by q_0 and that a temporary stack denoted temp_stack is globally available. The algorithm of Figure 4.4 depicts the body of a driver with actions deferred on one token symbol. Initially, the parser is in configuration $q_0 \mid \omega$ where q_0 is the start state, and ω is the whole input string. Starting with the initial configuration, the idea is to advance through the input stream one token at a time, executing all actions induced by the current input token at each step. Thus, the function *lookahead_action* is invoked at each step to check whether or not it is possible to advance. If so, the state stack is updated by replacing all of its topmost elements from pos+1 to #temp_stack by the corresponding temp_stack elements. Usually, pos+1=#temp_stack unless a non-empty sequence of empty reductions, each followed immediately by a goto action were executed prior to exiting the outer loop of the *lookahead_action* function. In such a case, #temp_stack exceeds pos+1 by the number of such empty reductions that were executed. If the valid action returned by the *lookahead_action* function is a shift-reduce action, then it and all its associated goto-reduce actions are processed. When the parser

```
state_stack := [$q_0$];
while true loop
    ppos := 0;    previous_stack := [ ];
    npos := 0;    next_stack := [ ];
    location_stack(#state_stack) := curtok.location;
    temp_stack := state_stack;
    act := lookahead_action(temp_stack, $t_1$, pos);
    while act $\neq$ error and act $\neq$ accept loop
        next_stack(npos+1..) := temp_stack(npos+1..);
        location_stack(pos+1..) :=
            [curtok.location : i in [pos+1..#next_stack]];
        if act is a shift-reduce action then
            top := #next_stack;
            until act is not a goto-reduce action loop
                top := top - RHS(act) + 1;
                act := GOTO(next_stack(top), LHS(act));
            end loop;
            next_stack(top+1..) := [act];
            pos := min(pos, top);
        end if
        act := lookahead_action(next_stack, $t_2$, npos);
        if act $\neq$ error then
            get next token;
            previous_stack(ppos+1..) := state_stack(ppos+1..);
            ppos := pos;
            state_stack(pos+1..) := next_stack(pos+1..);
            pos := npos;
        end if;
    end loop;
    if act =accept then
        return;
    end if
    error_recovery();
end loop;
```

Fig. 4.5: Driver with 3 deferred tokens

can advance successfully, the next token is read in and the process is repeated on the new configuration. If, on the other hand, the error action was returned by the *lookahead_action* function, the state stack is not updated and the error recovery routine is invoked instead.

It is not hard to see how this driver routine can be adapted to defer parsing actions on n tokens given n state stacks. In experiments with this method, parsing has been deferred for three tokens. The three stacks that are used are: previous_stack which captures the configuration of the parser prior to processing any action induced by *prevtok*, state_stack which captures the configuration prior to processing actions induced by *curtok*, and next_stack which captures the configuration prior to processing actions induced by the successor of *curtok*. Associated with each of these stacks are three integer variables: ppos, pos and npos which are used to mark the position of the top element in the corresponding stack that is still valid after the actions induced by the relevant lookahead symbol are applied. Figure 4.5 shows the body of a driver routine with actions deferred on three input symbols. This deferral usually increases the time requirement of the parser by 25% or less.

4.3 Recovery Strategies

Each recovery attempt is called a *trial.* The effectiveness of a recovery is evaluated using a validation function, *parse_check*, which indicates how many tokens in the input buffer can be successfully parsed after the repair in question is applied: *parse_check distance.* A recovery trial is not considered successful unless the parse_check distance is greater than or equal to a certain value, called *min_distance*. Experiments have shown that a good choice for *min_distance* is 2 [36].

The *parse_check* function is essentially an LR driver that simulates the parse until it has either shifted all the tokens in the buffer, completed the parse successfully, or reached a token in error. The same approach taken in implementing the *lookahead_action* function can be extended to implement the *parse_check* function; i.e., a temporary stack can be used to keep track of all state information related to transitions induced by the lookahead tokens in the buffer, thus, avoiding copying the state stack or destroying it.

In the remainder of this section, the implementation of the three recovery strategies mentioned earlier are described in detail.

4.3.1 Simple Recovery

Given a configuration: $q_1, q_2, \ldots, q_m \mid t_1, t_2, \ldots, t_n$, where t_1 is assumed to be the error token, the simple recovery finds the best possible *simple repair* (if any) for that configuration. The selection of a best simple repair is based on three criteria:

- the parse_check distance
- the *misspelling index*
- the order in which the trials are performed.

The misspelling index is a real value between 0.0 and 1.0 that is associated with each simple recovery trial. When a new token is substituted for the error token - a *simple substitution* - a misspelling function is invoked to determine the misspelling index; i.e., the relative proximity of the two tokens in question expressed as a probabilistic value. For other kinds of recoveries, the misspelling index is set to a constant value depending on the recovery in question and other conditions. This will be discussed later.

Simple recoveries are attempted in the following order: merging of the error token (t_1) with its successor (t_2); deletion of t_1; insertion of each terminal candidate in $t\_symbols(q_m)$ before t_1; substitution of each legal terminal candidate in $t\_symbols(q_m)$ for t_1; insertion of each nonterminal candidate in $nt\_symbols(q_m)$ before t_1; and, finally, substitution of each nonterminal candidate $nt\_symbols(q_m)$ for t_1; For now, one can assume that for a state q, $t\_symbols(q)$ and $nt\_symbols(q)$ yield the sets of all terminal and nonterminal symbols, respectively, on which actions are defined in q. Optimization of these sets is discussed in section 5.2.1.

As the trials are performed, the simple recovery routine keeps track of the most succesful trial. Initially, the merge recovery is chosen since it is attempted first. If a subsequent recovery yields a larger parse_check distance than the previously chosen recovery or it yields the same parse_check distance but with a greater misspelling index, then it is chosen instead as the best recovery candidate.

For the merge trial, the character string representation of t_2, is concatenated to the character string representation of t_1 to obtain a merged string s. A test is then performed to determine if s is the character string representation of some $t \in t\_symbols(q_m)$. If such an element t, called a *merge candidate*, is found, a new configuration is obtained by temporarily replacing t_1 and t_2 with t in the input sequence and the parse_check distance is computed for this new configuration.

As described in the previous section, the deferred driver insures that the state q_m on top of the stack of the error configuration is the state entered prior to the execution of any action on t_1. In that configuration, it may be possible to execute a sequence of reduce, goto-reduce and goto actions before the illegality of t_1 is detected in another state q_e. In such a case, the elements in $t\_symbols(q_m)$ that are also in $t\_symbols(q_e)$ are given priority in applying the insertion and substitution trials. (It is not hard to show that $t\_symbols(q_e) \subseteq t\_symbols(q_m)$.) This ordering is not crucial but its benefits can be seen in the following example:

```
write(1*5+6;2*3,4/2)
```

In this erroneous Pascal statement, a semicolon is used instead of a comma after the first parameter. Assume state q_m is the first state that encounters the semicolon. At that point, the parser has just shifted an expression operand and the set of valid lookahead symbols includes not only the comma but all the arithmetic operators. However, if the parser is allowed to interpret the operand as a complete expression, it will enter an error state q_e where the comma is the only candidate.

In order to give priority to the candidates in an error state q_e, it is necessary to identify when the parser has entered such a state. State q_e can be computed in the *lookahead_action* function by inserting the following statement after lines 3. and 14. in Figure 4.3:

```
error_state := act;
```

4.3.1.1 Misspelling Index

For a successful merge trial, the misspelling index is set to 1.0 since the merged string must perfectly match the character string representation of the merge candidate. This ensures that if a merge trial yields a successful recovery, any subsequent recovery that is applicable in the given context will not be chosen over the merge recovery unless it yields a longer *parse_check* distance. Consider, the following example:

```
if x >  = y then
   x :  = z;
```

In this case, it is most likely that the programmer inadvertently inserted a space character between ">" and "=" in the first line, and between ":" and "=" in the second line. Both errors can be successfully repaired with merging. However, from a syntactic point of view, the first error is just as easily repaired by deleting the ">" symbol (or the "="); but, since the merging trial is perfomed first and, in addition, it yields a higher misspelling index, it is chosen over the deletion.

As mentioned earlier, a misspelling function is invoked to calculate the misspelling index for a simple substitution. For all other recoveries, the misspelling index is set to 0.0 unless the candidate in question was identified as a special *end-of-line* terminal and the correction is to be made at the end of a line. In that case, the misspelling index is set to 1.0. (A boolean attribute is associated with each input token indicating whether or not it is located at the end of a line in the input). In Pascal, the end-of-line terminal is ";". Consider the following example:

```
x := y
p(x);
```

The error token in this incorrect fragment is p. Syntactically, any arithmetic operator, say "+", inserted after the symbol y would repair this error. However, it is clear from the context that since y is at the end of the line, ";" is a better candidate. Setting the misspelling index to 1.0 when inserting the end-of-line terminal at the end of a line gives it precedence over other insertion candidates.

4.3.1.2 Misspelling Function

In designing a misspelling function for syntactic error recovery, special attention should be paid to the kinds of errors that a programmer is likely to make. For example, the use of abbreviations, such as `int` instead of `integer` or `proc` instead of `procedure`, is common. Often, words that start with a similar prefix are substituted for each other: e.g., `procedure` for `program`. The algorithm used should be sensitive enough to identify such a pair of words as a misspelling error (though with a low probability); but, it should also be able to reject a pair of words such as `return` and `turner` that are permutations of the same set of characters but unlikely to be incorrect misspellings of each other.

The misspelling algorithm used in this method is an adaptation of an algorithm proposed by Jüergen Uhl [42]. In that approach, three kinds of misspelling errors are identified: transposition of two adjacent characters, mismatch of corresponding characters and omission (insertion) of a character. The idea is to scan the two strings simultaneously and compute the following information as they are being traversed.

```
--  The function misspell takes two arguments: s1 and
--  s2 which are character strings of arbitrary length.
function misspell(s1, s2);
    i := 0;
    j := 0;
    match_count := 0;
    prefix_length := 0;
    error_count := 0;
    while ((i < #s1) and (j < #s2)) loop
        if s1(i) = s2(j) then    -- matched characters?
            match_count := match_count + 1;
            i := i + 1;
            j := j + 1;
            if error_count = 0 then    -- prefix character?
                prefix_length := prefix_length + 1;
            end if;
        elseif s1(i+1) = s2(j) and s1(i) = s2(j+1) then    -- transposition?
            match_count := match_count + 2;
            i := i + 2;
            j := j + 2;
            error_count := error_count + 1;
        elseif s1[i+1] = s2[j+1] then    -- mismatch?
            i := i + 1;
            j := j + 1;
            error_count := error_count + 1;
        else -- definitely a deletion!
            if (#s1-i) > (#s2-j) then    -- suffix of s1 longer?
                i := i + 1;
            elseif (#s2-j) > (#s1-i) then    -- suffix of s2 longer?
                j := j + 1;
            else
                i := i + 1;
                j := j + 1;
            end if;
            error_count := error_count + 1;
        end if;
    end loop;
    if (i < #s1) or (j < #s2) then
        error_count := error_count + 1;
    end if;
    if error_count <= (min(#s1, #s2) / 6 + 1) then    -- check error threshold
         pattern_count := prefix_length;
    else pattern_count := match_count;
    end if;
    return float(pattern_count) / (max(#s1, #s2) + error_count);
end misspell;
```

Fig. 4.6: Misspelling function

- the length of the initial prefix of the two strings that matches (prefix_length)
- the total number of characters that match (match_count)
- the number of errors found (error_count)

With this information, the probability that a string s_1 is a misspelling of a string s_2 is calculated as follows. Let the error threshold be the length of the shortest string divided by six plus one; i.e., at least one error is allowed plus an additional error for each six characters in the shortest string. If the number of errors detected by the pattern match (error_count) is less than or equal to the error threshold then match_count represents the result of the pattern match (pattern_count); otherwise, only the number of initial characters that matched (prefix_length) are considered. The final result is obtained by dividing pattern_count by the length of the longest string plus error_count. Figure 4.6 shows a complete implementation of this algorithm.

Certain static characters are also likely to be substituted for others. For example,

$$\begin{array}{ccc} ; & \leftrightarrow & , \\ ; & \leftrightarrow & : \\ . & \leftrightarrow & , \\ ' & \leftrightarrow & " \end{array}$$

In this method, the characters of each of the above pairs are considered to be 0.33% likely to be a misspelling of each other.

4.3.2 Phrase-Level Recovery

Phrase-level recovery is based on the identification of an *error phrase* which is then deleted from the input or replaced by a suitable nonterminal symbol or *reduction goal.* If the string:

$$q_1, \ldots, q_m \quad | \quad t_1, \ldots, t_n \tag{4.1}$$

is an error configuration, then a substring

$$q_{i+1}, \ldots, q_m \quad | \quad t_1, \ldots, t_{j-1} \tag{4.2}$$

$1 \leq i \leq m$, $1 \leq j \leq n$, of that configuration is an error phrase (of the configuration) if removing that substring allows the parser to advance at least *min_distance* tokens into the forward context, or if there is a nonterminal A such that a valid action is defined in state q_i on A, and after processing A, the parser can advance at least *min_distance* into the forward context. Here, q_i, A and t_j are the *recovery state*, *reduction goal* and *recovery symbol*, respectively.

In [27], the authors present a detailed discussion of different strategies that are used for selecting error phrases, and the advantages and disadvantages of each strategy. In general, the search for the error phrase begins with the shortest possible error phrase; that is, with the one consisting only of the vertical bar, and proceeds with larger and larger segments of the configuration. A search order may be used that consumes state stacks faster than input symbols or consumes input symbols faster than state stacks or a more balanced scheme may be used.

The scheme used in this method to select error phrases reflects a fundamental distinction that is made among three different kinds of errors. Consider the error configuration (4.2) above. The case of the empty error phrase is considered during simple recovery as a nonterminal insertion. Similarly, the case of an error phrase $\varepsilon|t_1$ being deleted or replaced by a nonterminal candidate is processed by a simple deletion or substitution. Next, priority is given to a successful phrase-level recovery that consumes no input symbol and requires no insertion of a reduction goal; i.e., a recovery based on the removal of an error phrase of the form $\beta|\varepsilon$ where $\beta \neq \varepsilon$. This kind of error is called a *misplacement error*, and β is called a *misplaced phrase*. The following Pascal example illustrates this case:

```
1. program P(INPUT,OUTPUT);
2.     var I:real;
       <--------->
***Error: Misplaced construct(s)
3.     type ORDER=array[1..MAX] of real;
4.     var Q:integer;
5. begin
6. end.
```

Finally, the case in which one or more input symbols and/or states must be deleted or replaced with a nonterminal candidate is considered. In that case, input symbols are consumed faster than states. In other words, the error phrases are selected as indicated by the row-major order of the table below:

$$\begin{array}{ccc}
\varepsilon|\varepsilon & \cdots & \varepsilon|t_1,\ldots,t_n \\
q_m|\varepsilon & \cdots & q_m|t_1,\ldots,t_n \\
\vdots & & \vdots \\
q_2,\ldots,q_m|\varepsilon & \cdots & q_2,\ldots,q_m|t_1,\ldots,t_n
\end{array}$$

In this final case, each error phrase selected is removed from the base configuration (4.1). An initial attempt is made to recover by parse checking the resulting configuration. This action, called *phrase deletion*, can be viewed as a multiple deletion of the symbols that make up the error phrase. Next, each element in the set of nonterminal candidates for the newly exposed state on top of the state stack is substituted, in turn, for the error phrase and the *parse_check* function is invoked to determine its viability. This action is called a *phrase substitution*. This process continues until a successful recovery is found or all the possibilities are exhausted.

In phrase-level recovery, the aim is to find a repair that least alters the original configuration. For this reason, misplacement trials are performed separately from the other phrase-level trials and given higher priority, since such a repair does not delete any symbol from the forward context and tends to remove whole structures from the left context that have been previously analysed. The *parse_check* distance is used as the criterion to select the best misplacement repair. After the misplacement trials, a phrase deletion and substitution trial is performed on successive error phrases. The selection of a best deletion or substitution repair is based on the length of the relevant error phrase and the *parse_check*

```
if_stmt      →   if cond then
                       st_list elsif_list opt_else
                 end if ;
st_list      →   stmt | st_list stmt
elsif_list   →   ϵ | elsif_list elsif cond then st_list
opt_else     →   ϵ | else st_list
stmt         →   . . . | if_stmt | . . .
```

Fig. 4.7: BNF rule for Ada **if** statement

distance, with deletion having priority over substitution in case of a tie. The length of an error phrase $\beta|x$ is obtained by adding the length of the string x to the number of non-null symbols in β.

Given the best misplacement repair and the best deletion or substitution repair, if the misplacement repair is based on a shorter error phrase or it yields a longer *parse_check* distance, then it is chosen. Otherwise, the deletion or substitution is chosen.

4.3.3 Scope Recovery

One of the most common errors committed by programmers is the omission of block closers such as an **end** statement or a right parenthesis. Such an error is referred to as a *scope error.* Scope errors are common because the structures requiring block closers are usually recursive structures that, in practice, are specified in a nested fashion. In such a case, a matching block closer must accompany each structure in the nest. For example, if a user specifies an expression that is missing a single right parenthesis, simple recovery can successfully insert that symbol. However, if two or more right parenthesis are missing, neither simple nor phrase-level recovery can successfully repair such an error. Similarly, consider the BNF rule for an Ada *if-statement* in Figure 4.7 [28]: If an Ada *if-statement* is specified without the "**end if**;" closer, neither of the two recovery techniques mentioned so far can effectively repair this error. The repair that is necessary for this kind of error is the insertion of a sequence of symbols, called *multiple symbol insertion.*

Scope recovery was first introduced by Burke and Fisher [36]. Their technique requires that each closing sequence be supplied by the user as a list of terminal symbols. Scope recovery is attempted by checking whether or not the insertion of a combination of these closing sequences can allow the parser to recover.

By contrast, the scope recovery technique used in this method is based on the identification of one or more recursively defined rules that are incompletely specified, and insertion of the appropriate closing symbols to complete these phrases. All necessary scope information required by this method is precomputed automatically from the input grammar. In addition, because the method is based on a pattern match with complete rules rather than just the insertion of closing sequences of terminal symbols, the diagnosis of scope errors is more accurate in that it identifies whole structures that are incompletely specified instead of just the missing sequence of closing terminals.

4.3.3.1 Scope Information

Definition 4.3.1: A rule $A \rightarrow \alpha B \beta$ is a **scoped rule** if and only if the following conditions are satisfied:

1. $B \Rightarrow^* \gamma A \delta$, for some arbitrary strings γ and δ
2. $\beta \not\Rightarrow^* \varepsilon$
3. $\alpha \not\Rightarrow \varepsilon$ or $B \not\Rightarrow^*_{lm} A\psi$, for some string ψ
4. $\alpha \neq \epsilon$ or ($\exists$ a viable prefix $\phi A \mid$ either ϕB is not a viable prefix or $\nexists C \in N$ such that $\phi C \Rightarrow^*_{rm} \phi A$ and $\phi C \Rightarrow^+_{rm} \phi B$).

Condition 1 simply states that for a rule to be a *scoped rule* it must be recursive. Condition 2 guarantees that the closing sequence following the recursive symbol in the right-hand side of the rule is not nullable. These two conditions are consistent with the intuitive notion of a scope presented earlier. By contrast, the purpose of conditions 3 and 4 is to exclude some unnecessary cases. For example, consider the following left-recursive rule:

$$\mathsf{List} \rightarrow \mathsf{List}\ ,\ \mathsf{atom}$$

Condition 3 eliminates such a rule from consideration since the initial prefix preceding the recursive nonterminal List in the right-hand side is empty and List $\Rightarrow^*_{lm}$ List. The rules affected by condition 4 are more complex. Consider the following grammar:

List	→	Sublist
List	→	Sublist List
Sublist	→	atom
Sublist	→	(List)

Since Sublist $\Rightarrow$ (List), the right-recursive rule List → Sublist List satisfies conditions 1, 2 and 3. However, it does not satisfy condition 4, because List $\Rightarrow^+_{rm}$ Sublist. In essence, condition 4 eliminates from consideration all rules of the form $A \rightarrow B\beta$ such that whenever A is introduced through closure, it is introduced by some nonterminal C (where C may be the symbol A itself, as in the above example) which can produce both A and B by a sequence of right-most derivations.

In the above example, the rule Sublist → (List) is a scoped rule. In the example of Figure 5.5, the rule $P \rightarrow (E)$ is a scoped rule since P can be derived from E. The if_stmt rule of Figure 4.7 is also a scoped rule since each of the symbols: st_list, elsif_list and opt_else in that rule can recursively derive a string containing the symbol if_stmt. A *scope* can be derived from a scoped rule for each recursive symbol in the right-hand side of the scoped rule.

A scope is a quintuple (π, σ, a, A, Q) where π and σ are strings of symbols called *scope prefix* and *scope suffix*, respectively, a is a terminal symbol called the *scope lookahead*, A is a nonterminal symbol called the *left-hand side* and Q is a set of states. The scope prefix is the prefix of a *suitable string* derivable from the scoped rule in question. It is used to determine whether or not a recovery by the associated scope is applicable; i.e., at run time, a repair by a given scope is considered only if this initial substring of the suitable string can be successfully derived before the error token causes an error action. The scope suffix is the suffix (of the suitable string) that follows the scope prefix. When diagnosing a scope error, the user is advised to insert the symbols of the scope suffix into the input stream to complete the specification of the scoped rule. The scope lookahead symbol (string, if the grammar is LR(k), $k > 0$) is a terminal symbol (string) that may immediately follow the prefix in a legal input. The left-hand side of the scope is the nonterminal on the left of the scoped rule. The set Q contains the states of the LR(k) automaton in which the left-hand side can be introduced through closure.

Given a scoped rule $A \to \alpha B \beta$, the scope information related to B is computed as follows. Since $\beta \not\Rightarrow^* \varepsilon$, there exists a string $\psi X \phi$ such that $\beta \Rightarrow^* \psi X \phi$, $\psi \Rightarrow^* \varepsilon$, and $X \Rightarrow^*_{rm} a\omega$, for some string ω. Let $\alpha B \psi X \phi$ be the suitable string mentioned above, then a valid *scope* for the rule $A \to \alpha B \beta$ is $(\alpha B \psi, X\phi, a, A, Q)$, where Q is the set of states in the LR automaton containing a transition on A.

As an example, consider the if_stmt rule of Figure 4.7 and the scope induced by the nonterminal st_list in its right-hand side. To put it in the form $A \to \alpha B \beta$, let B be the symbol "st_list". It follows that α is the string "**if** cond **then**", and β is the string "elsif_list opt_else **end if**;". Let ψ be the string "elsif_list opt_else" and let X be the symbol "**end**". One observes that β is exactly in the desired form $\psi X \phi$. Thus, assuming the set of transition states Q is available, the scope induced by st_list for the if_stmt rule is:

(**if** cond **then** st_list elsif_list opt_else, **end if**;, **end**, if_stmt, Q)

The other recursive symbols in the if_stmt rule - elsif_list and opt_else - induce exactly the same scope as st_list, since they are both *nullable*.

4.3.3.2 Scope Error Detection

Given an error configuration:

$$q_1, \ldots, q_m \quad | \quad t_1, \ldots, t_n$$

and a set of scopes:

$$\{(\pi_1, \sigma_1, a_1, A_1, Q_1), \ \ldots, \ (\pi_l, \sigma_l, a_l, A_l, Q_l)\},$$

the applicability of scope recovery to this configuration is determined as follows. For each scope $(\pi_i, \sigma_i, a_i, A_i, Q_i)$, a three-step test is performed:

step 1: The *lookahead_action* function is invoked with a_i as the current token to check if a_i is a valid lookahead symbol for the viable prefix. As a side-effect, this function updates the state stack configuration (using a temporary stack) to reflect all reduce actions, including empty reductions, induced by a_i. If the action returned by *lookaheadaction* is the error action then the whole test fails. Otherwise, assuming, once again, that m denotes the upper bound subscript of the updated state sequence, step 2 is executed.

step 2: A pattern match is made between the prefix π_i and the topmost $|\pi_i|$ symbols of the viable prefix, i.e., the string obtained from the concatenation of the in_symbols of the states: $q_{m-|\pi_i|+1}..q_m$. Again, if this test fails, the whole test fails. Otherwise the final step is executed.

step 3: If $q_{m-|\pi_i|} \in Q_i$ then the test succeeds. Otherwise, the test fails.

If the three-step test is successful, then a parse check is performed on the configuration: $q_1, \ldots, q_{m-|\pi_i|}, q_A \mid t_1, \ldots t_n$, where q_A is the successor state of q_m and A[1]. If the *parse_check* function can parse at least *min_distance* symbols, the scope recovery is successful. Otherwise, it is invoked recursively with the new configuration above and the process is repeated until scope recovery either succeeds, or there are no more possibilities to try.

When scope recovery is successful, the sequence of scopes that resulted in the successful recovery must be saved for the issuance of an accurate diagnostic.

Figure 4.8 depicts a complete implementation of the scope error detection algorithm. The algorithm mirrors the preceding discussion in a straightforward manner except for the tests in line 12 and the code segment in lines 2 through 5. In line 12, the test for top > 0 ensures that the stack is longer than the scope prefix with which it is being matched and the test for top < min(#stack, pos+1) prevents the algorithm from considering a scope whose prefix would match a null string. This latter test also guarantees that each time a scope is tested and applied it does not extend the length of the stack. The code segment in lines 2 through 5 prevents the algorithm from visiting a given configuration more than once. Since the application of a scope never extends the state sequence it started with, one only needs to keep track of the states that have been entered at each index position of the initial state sequence.

The emphasis in writing the code of Figure 4.8 was on the clarity of the exposition rather than efficiency. In particular, note that stack should not be copied into scope_stack(as in line 7) for each application of a scope. Instead, the attempt to match a scope prefix with the viable prefix should be made using the relevant segment of the state sequence that is in stack and the relevant segment that is in temp_stack. Since the application of a scope never extends the stack, one can simply substitute stack for scope_stack in the recursive call on line 19.

4.3.3.3 User-Supplied Scopes

This scope recovery method can be extended to accommodate any kind of multiple insertion of symbols by allowing the user to specify his own "scopes". Scopes are explicitly specified with productions that use the special error symbol, %*error*, as a right-hand side marker to separate the scope prefix from its suffix. These productions are called *user-supplied scoped rules* and the scopes derived from them are called *user-supplied scopes.*

The parser generator upon scanning a rule with %*error* in its right-hand side identifies such a rule as a scope whose prefix consists of the symbols preceding %*error* and whose

[1] If the action in q_m on A is a goto-reduce, the parser is simulated through the whole sequence of goto-reduce actions that follow, until a goto action is encountered. This final goto is executed and the resulting state sequence is used instead. Note that these actions do not consume any input symbol.

```
-- Let scope_sequence and state_seen be global variables. scope_trial is invoked with the sequence
-- of states: stack=$q_1, \ldots, q_m$ in the error configuration. The input sequence $t_1, \ldots, t_n$ is assumed
-- to be global. Initially, scope_sequence=[ ] and state_seen=[$\emptyset$ : i $\in$ [1..$m$]].

proc scope_trial(stack);
    if ($q_1$ := stack(#stack)) $\in$ state_seen(#stack) then
        return;
    end if;
    state_seen(#stack) := state_seen(#stack) $\cup$ $\{q_1\}$;
    for each scope $(\pi_i, \sigma_i, a_i, A_i, Q_i)$ loop
        scope_stack := stack;
        act := lookahead_action(scope_stack,$a_i$,pos)
        if act $\neq$ error then
            scope_stack(pos+1..) := temp_stack(pos+1..);
            top := #scope_stack - $|\pi_i|$;
            if top > 0 and top < min(#stack, pos+1) then
                pref := [in_sym(scope_stack(j)) : j in top+1..#scope_stack];
                if pref = $\pi_i$ and scope_stack(top) $\in$ $Q_i$ then
                    until act is a not goto-reduce action loop
                        top := top - RHS(act) + 1;
                        act := GOTO(scope_stack[top], LHS(act));
                    end loop;
                    if parse_check(scope_stack(1..top)+[act], $t_1, \ldots, t_n$) > min_distance then
                        scope_sequence := [i];
                        return;
                    else
                        scope_trial(scope_stack);
                        if scope_sequence $\neq$ [ ] then
                            scope_sequence := scope_sequence + [i];
                        end if;
                        return;
                    end if;
                end if;
            end if;
        end if;
    end loop;
end scope_trial;
```

Fig. 4.8: scope_trial procedure

suffix consists of the symbols following it. The symbol *%error* is used as the lookahead for the scope but it is neither included in the prefix nor the suffix. Therefore, a single rule should never be used to specify more than one user-supplied scope because that will cause the suffix of some scopes to contain the special *%error* symbol.

When specifying scopes, special attention must be paid to the kinds of symbols that are likely to be omitted by a programmer. For example, consider an Ada record_type_definition and the context in which it is used:

```
full_type_declaration     →  type identifier [discriminant_part] is type_definition ;
type_definition           →  ... | record_type_definition | ...
record_type_definition    →  record
                                 component_list
                             end record
```

Even though the above rule for record_type_definition is terminated by an "**end record**" sequence, it is not a scope, because in Ada, no string containing record_type_definition can be derived from component_list. However, simply duplicating the record_type_definition rule to identify "**end record**" as a scope closer is unlikely to be effective since a programmer who omits this sequence would, most likely, also omit the terminating semicolon. In order to identify the complete closing sequence one must respecify the whole full_type_declaration for a record_type_definition as follows:

```
full_type_declaration    →  type identifier [discriminant_part] is
                            record
                                component_list
                                %error
                            end record ;
```

During normal parsing, user-supplied scoped rules are never reduced except when used for scope recovery because *%error* is only used internally the error recovery procedure and is never recognized as an input symbol by the lexical analyzer. When detecting scope errors and applying scope recovery at run time, user-supplied scopes are treated in the same manner as the automatic ones.

4.4 Recovery Phases

This section describes how the different error recovery strategies discussed in the previous sections are incorporated into the unified two-phase scheme of this method. Error diagnosis, error repair and a suitable data structure for implementing and managing the input stream are also discussed.

It is assumed that the driver with 3 deferred tokens of Figure 4.5 is used. This driver can detect an error either on *curtok* or on the successor of *curtok* which will be denoted *succtok*. When the parsing starts (or restarts after a recovery), if *curtok* is in error, the parser stops immediately and invokes the error recovery routine with a single configuration, namely, the configuration prior to the execution of any action on *curtok*. The state sequence of this error configuration is contained in state_stack. If the parser can successfully process *curtok* but fails on *succtok*, the error recovery routine has access to both the *curtok* configuration and the configuration prior to the execution of any action on *succtok*. The state sequence of this configuration is contained in *next_stack*. If the parser is able to parse at least 2 tokens successfully before an error is detected, the error is detected on *succtok*, and in addition to the *curtok* and *succtok* configurations, the configuration of the parser prior to the execution of any action on *prevtok* is also available. The state sequence of this configuration is contained in *previous_stack*.

At the global level, the effectiveness of a recovery trial is measured based on two criteria:

- the number of symbols that must be deleted if the repair in question is applied
- the *parse_check* distance of the recovery

The primary phase recovery which includes all recovery trials that are based on at most a single input token modification is attempted first. If a successful primary phase recovery is found that cannot be beaten by any other recovery in terms of the criteria above, it is accepted. If such a primary phase recovery is not found, secondary phase recovery is attempted. If a successful secondary phase recovery is found, then it is accepted. Otherwise, the error recovery gets into a form of "panic mode", where the current input buffer is flushed, new input tokens are read in and secondary phase recovery is attempted again. This process is repeated until either a successful phrase-level recovery is obtained or the end of the input stream is reached.

When the error recovery procedure finds a successful recovery, it issues a diagnosis for it and repairs the configuration before returning control to the parser.

4.4.1 Primary Phase

In the primary phase, error recovery is applied on each available configuration, starting with next_stack, proceeding with state_stack and finally processing previous_stack.

For each configuration, scope recovery is attempted first followed by simple recovery. The same criteria used in choosing a simple recovery is used in the primary phase. The misspelling index of a scope recovery trial is set to 1.0. Thus, for a given configuration, a successful scope recovery always has priority over a simple recovery trial that yields the same parse_check distance.

If a successful recovery is obtained from the primary phase and its stack configuration is next_stack or state_stack, the recovery trial is evaluated against certain phrase-level recovery trials on the stack configuration in question before being accepted. The phrase-level recovery trials in question are the ones whose repair actions would have as little impact on the recovery configuration as a simple recovery. They are misplacement recovery trials and scope recovery trials that require the deletion of a single input token. The idea is to ensure that none of these borderline recoveries can be more effective than the best primary phase recovery.

4.4.2 Secondary Phase

In the secondary phase, phrase-level recovery is applied first on the next_stack configuration if it is available and then on the state_stack configuration. Phrase-Level recovery is never attempted on the previous_stack configuration, because it is not always possible to issue an accurate diagnosis for a recovery found in that configuration (see Section 4.4.3) or to properly repair that configuration (see Section 4.4.5).

If a successful phrase-level recovery is obtained, a check is made to see if the error can be better repaired by the closing of some scopes followed by less radical surgery. Consider the following Pascal example:

```
if count[listdata[sub] := 0 then
    x := (( 3 ]];
```

In the first line, the programmer is missing a closing "`]`" and the assignment operator "`:=`" is used instead of a relational operator. This error is detected on the symbol "`:=`". In the second line, the programmer used two "`]`" instead of "`)`" to close a parenthesized expression and the error is detected on the first "`]`". Nothing short of a phrase deletion of the sequence "`[listdata[sub] := 0`" in the first instance and a phrase substitution of "expression" for the sequence "`(( 3 ]]`" would successfully repair these errors. However, it is not difficult to see that they can be repaired more accurately, using scope recovery by proceeding as follows. Before accepting a phrase-level recovery based on an error phrase $\beta|x$, a scope recovery check is performed on the recovery configuration, followed by the deletion of up to $|x|$ tokens in the right context. If the scope recovery is successful, then its associated repair actions are applied without the subsequent deletion and the secondary phase returns successfully. The parser fails right away and once again invokes the error recovery procedure. On this next round, primary and secondary phase recovery are attempted again. This subsequent attempt will at best fix the remaining input or at worst delete x from the input. In the example above, the missing "`]`" is inserted and "relational_operator" is substituted for "`:=`" in the first line. In the second line, two closing "`)`" are inserted, followed by a deletion of the pair "`]]`" (See figure 4.2).

4.4.3 Error Diagnosis

In order to accurately diagnose an error, one must identify the location of the tokens that are in error. This is straightforward for a simple recovery since such a recovery involves the modification of one or two input tokens and the location of each token is available. Recall that each state in the state stack is associated with the location of the first token on which an action was executed in that state. Thus, given an error configuration $q_1, \ldots, q_m \mid t_1, \ldots, t_n$, if a successful phrase-level recovery based on an error phrase $q_{i+1}, \ldots, q_m \mid t_1, \ldots, t_{j-1}$ of that configuration is found, the location of the first token of this error phrase is the location associated with the recovery state q_i. If the state sequence associated with the error configuration is state_stack then the symbol t_1 is *curtok*. In that case, if $j = 1$, t_0 is *prevtok*. Similarly, if the state sequence is next_stack and $j = 1$ then $t_1 = \mathit{succtok}$ and $t_0 = \mathit{curtok}$. Finally, if the state sequence is previous_stack and $j = 1$ then $t_1 = \mathit{prevtok}$ and t_0 is undefined. Therefore, as mentioned earlier, if phrase-level recovery is attempted on the previous_stack configuration, one cannot identify the last symbol of an error phrase that does not contain any input symbol unless the predecessor of the previous token is also kept. When a scope recovery associated with a scope (π, σ, a, A, Q) is applicable on an error configuration like the one above, it can be viewed as a secondary substitution of the nonterminal A for the error phrase $q_{i+1}, \ldots, q_m \mid \varepsilon$, where $|\pi| = m$. That is, the location of the first token of this error phrase is the location associated with q_i

and the location of the last token is the location of t_0 as described above. (Note that this implies that the location of the last symbol in a scope error phrase of the previous_stack configuration cannot be accurately determined.)

Once the location of the error token or error phrase has been identified, an error diagnosis message is issued describing the repair. The diagnosis of a simple recovery repair is straightforward except that for an insertion or substitution some preprocessing of the repair token is required before the message is emitted. This will be discussed later. To diagnose a phrase deletion, the user is advised to delete the symbols in the error phrase in question. For a phrase substitution, if the relevant reduction goal is a nullable nonterminal, the diagnosis is treated like a phrase deletion. Otherwise, the reduction goal is suggested as a replacement for the error phrase. To diagnose a scope recovery, the user is advised to insert the symbols of the scope suffix in question after t_0, if the location of t_0 is defined, or before t_1, otherwise. In addition, it is very useful to identify the starting location of the scope which can be computed from the recovery state.

The main goal of an error message is to inform the programmer as to how the input source was modified. However, it is also desirable to issue error messages that give the user an accurate diagnosis of the error. In particular, whenever a symbol X is inserted into the text or it is substituted for an error token (by simple recovery) or for an error phrase (by phrase-level recovery), the symbol reported in the message is the *highest-level symbol* Y that can subsume X. That is, a symbol Y is used instead of X, if assuming ϕ is the viable prefix corresponding to the state sequence of the error configuration and w is the remaining input, $\phi Y w \Rightarrow^*_{rm} \phi X w$ and $\forall B \in N$ such that $B \neq Y$, $\phi B w \not\Rightarrow^+_{rm} \phi Y w$. Consider the following erroneous Pascal statement:

```
if n <= then POWER := else POWER := 2;
```

This statement is missing a subexpression after "<=" and an expression after the first ":=". Since simple recovery is attempted first on terminal symbols and any of the symbols: identifier, NIL, string_literal, integer_literal or real_literal can be reduced to a subexpression, one such symbol will be inserted into the text to repair each of these two errors. Depending on the semantic context, the arbitrary insertion of a symbol can be misleading even though it is syntactically correct. For example, it is clear from the example above that POWER is not a pointer variable. Nonetheless, NIL is a valid candidate that may be inserted after the first ":=". In reporting such an error, if the highest-level symbol associated with the repair candidate is used, simple_expression will be suggested as an insertion candidate after "<=" and expression will be suggested as an insertion candidate after the first ":=". (See Pascal BNF definition in [11].)

The computation of the highest-level symbol is straightforward. Starting in the recovery state one simply simulates the steps of the parser until a state q is entered where the candidate X can be shifted. Next, starting with state q as the only state in a stack and using the next input symbol t as lookahead, X is shifted and all reduce actions induced by t and their associated goto actions are applied until the parser wants to shift t or reduce below q. At that point, the last symbol on which a transition was made in q is the highest-level symbol that subsumes X. Figure 4.9 is an implementation of this algorithm.

```
function highest_symbol(stack, X, t);
    hs := X;
    q := stack(#stack);
    if $X \in N$ then
        act := GOTO(q, X);
    else if (act := ACTION(q, X)) is not a shift or shift-reduce action then
        act := lookahead_action(stack, X, pos);
        q := temp_stack(#temp_stack);
    end if;
    temp_stack := [q];
    if act is a shift or goto action then
        temp_stack := temp_stack + [act];
        act := ACTION(act, t);
    end if;
    top := 1;
    while act is a reduce action loop
        until act is not a goto-reduce action loop
            top := top - RHS(act) + 1;
            if top < 1 then
                return hs;
            elseif top = 1 then
                hs := LHS(act);
            end if;
            act := GOTO(temp_stack(top), LHS(act));
        end loop;
        temp_stack(top+1) := act;
        act := ACTION(act, t);
    end loop
    return hs;
end highest_symbol;
```

Fig. 4.9: highest_symbol function

4.4.4 Error Repair

A repair is applied by resetting the components of the main configuration: the input buffer and state_stack.

The resetting of the input buffer involves the insertion of some symbols into the buffer, the reading of new input tokens into the buffer, or the replacement of some buffer elements. A good scheme for managing the input buffer is discussed in the next section.

To reset state_stack, the first step is to initialize it with the proper state sequence if the successful recovery was found on previous_stack or next_stack. For a simple recovery, nothing more is required. For a phrase-level recovery, all states following the recovery state are removed from the stack. Similarly, for a scope recovery, the sequence of states on top of the stack that corresponds to the prefix of the scope in question is removed and the repair proceeds as if the error was a simple insertion of the left-hand side of the scope.

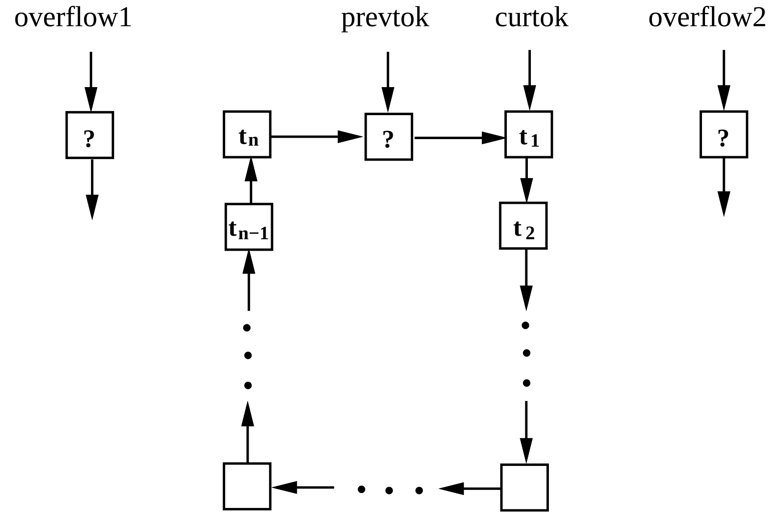


Fig. 4.10: Initial buffer configuration

4.4.5 The Input Buffer

The input buffer can be implemented as a fixed-size circular queue containing the previous token and the next n tokens to be processed, for some fixed integer n. *prevtok* is a pointer variable that identifies the previous token element in the circular queue. Its successor in the queue is identified by *curtok* which is also a pointer variable. Initially, the *prevtok* element is undefined and the next n tokens from the input are read into the remaining elements. See figure 4.10.

If the input is correct, the buffer is processed as follows. Each time the parser executes a shift (or shift-reduce action), the next input token is read into the *prevtok* element, *prevtok* is set to *curtok* and *curtok* is updated to point to its successor. This process continues until the parse terminates. (When the end of the input source is reached, the lexical analyser is expected to keep returning $\perp$.)

Note how this scheme easily accomodates the general case of an LALR(k) parser with variable lookahead-strings described in the previous chapter. If the parser needs to perform more lookahead, after having consulted a given element of the buffer, it obtains the next lookahead symbol by simply moving to the next element in the queue. Of course, this implies that the size of the input buffer queue must be greater than or equal to k.

When an error is encountered, the reparation of the configuration usually involves some modification of the input buffer. In particular, it may involve the insertion of some symbols. If a repair calls for the insertion of a symbol in front of *curtok* or *prevtok*, the buffer *overflows*. To accomodate such a repair, two extra *overflow* elements are needed. They are identified by two pointers: *overflow1* and *overflow2*. Let's consider the worst case, where the buffer contains a sequence: $t_0, t_1, \ldots, t_n$, where t_0 is the previous token, and the repair modification is to insert a symbol X in front of t_0. In that case, the previous token t_0 is copied into the *overflow2* element whose successor is set to be *curtok*; the new symbol X is placed in the *overflow1* element whose successor is set to be *overflow2* and

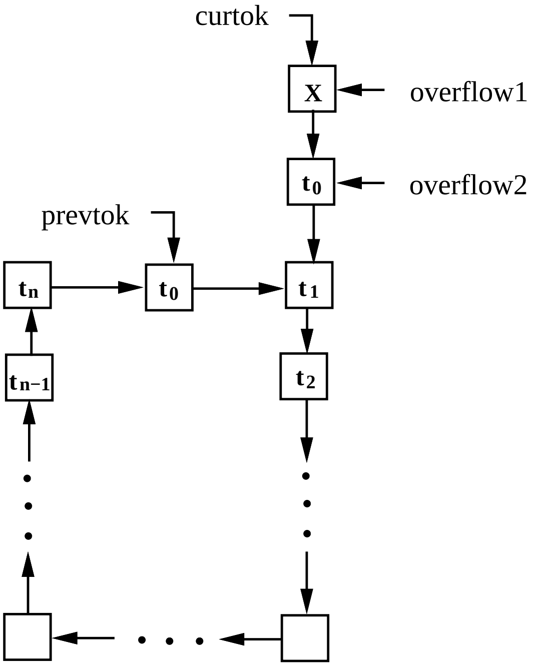


Fig. 4.11: Buffer configuration after insertion of t in front of t_0

curtok is set to *overflow1*. Figure 4.11 illustrates this case.

When executing a shift action on an overflow element, the parser simply updates *curtok* to point to its successor and no new tokens are read in. When the parser reenters the queue, the processing continues as before, and the next shift action updates the previous pointer. Note that this approach assumes that no error will occur on an overflow element. For normal recoveries, this can be guaranteed by *min_distance* being set to a value greater than or equal to 2. However, this restrinction explains the main reason why secondary phase recovery is not attempted on the previous_stack configuration. Recall from section 4.4.2 that in some cases, instead of applying a phrase-level recovery, a scope recovery is applied and control is returned to the parser with a configuration that will declare an error on the next input symbol. In such a case, if previous_stack is the state sequence that is used, then after the repair is applied, the input buffer is placed in a configuration similar to the configuration of Figure 4.11, where the symbol X is the left-hand side of the scope. Thus, after processing X, the parser will detect an error on an overflow element.

There are other repair cases where one or both overflow elements are needed in the resetting of the input buffer. The detail of these cases is left to the reader.

4.5 Remarks

This error recovery method has been successfully implemented. Parsers were built for Ada and Pascal and tested on the Ada examples of [36] and the Pascal examples of [20]. (Appendix E shows some of these results.) Penello and DeRemer [17] proposed that the quality of a repair be rated "excellent" if it repaired the test as a human reader would have, "good" if it resulted in a reasonable program and no spurious errors, and "poor" if it resulted in one or more spurious errors. Based on these categories, the performance of this method on the test set of [20] was 85.9% excellent, 14.1% good and 0.0% poor. In fact, most of the "good" recoveries resulted from errors whose repair required some kind of semantic judgement. The time performance of this method is excellent, usually requiring less than 50 milliseconds per error on a 16 MHz PS/2 model 80.

5. TABLE OPTIMIZATION AND GENERATION

In general, one of the main objectives that must be achieved in generating parsing tables is that they must be time- and space-efficient. In addition, if the resulting parser must recover from input errors, the tables generated must retain enough important information about the input grammar and its LR(0) automaton to accommodate error recovery. Unfortunately, these two goals are sometimes conflicting. For example, as mentioned earlier, when space-saving optimizations such as default reductions (and others that will be described in this chapter) are used, the resulting parsing tables lose certain information about the automaton. Space-saving optimizations are necessary in order to reduce the parsing tables to an acceptable size but the information they lose (such as the viable candidates in a given state) is usually important to insure good recoveries.

In the past, authors who have studied LR error recovery have proposed that some tradeoffs be made regarding space-efficiency, time-efficiency and quality of recoveries. For example, in [21], it is suggested that default reductions not be used in certain states to prevent premature reductions in these states and that only terminal symbols on which transitions are defined in a given state be considered as simple recovery candidates (this information must be available in the parsing tables). In [36], a deferred parsing technique using two parsers is used to avoid premature reductions and all terminal symbols are considered as possible candidates during simple recovery; in effect, time-efficiency is sacrificed in order to gain space-efficiency and obtain good recoveries.

By contrast, the approach taken in this method is to treat the issue of table compaction and optimization for parsing separately from the issue of table compaction and optimization for error recovery. The data for each activity is aggressively optimized and compacted, separately. The result is that the amount of space used for each of these two sets of tables is very small, (usually, the total amount of space used is much smaller than it would be for a single set of parsing tables in other methods). Nonetheless, these tables are very time-efficient and no useful error recovery information is lost.

5.1 Parsing Tables

As described earlier, an LALR(k) parser with varying-length lookahead strings can be represented by a pair of parsing tables in matrix form: ACTION and GOTO. Each row of a parsing table is labeled with a distinct state of the parser. Each column of ACTION is labeled with a terminal symbol and each column of GOTO is labeled with a nonterminal symbol. The entry in a parsing table for a given row and column is the parsing action associated with the corresponding state-symbol pair. In essence, the matrix is used to store the characteristics of the pushdown *automaton* that obeys the grammatical rules of the language. In terms of speed performance, the matrix is an efficient method for representing an LR parsing table. Unfortunately, such a matrix is usually very sparse, requiring an excessive amount of space relative to the number of useful entries it contains. For example, an LALR(1) parser for the Ada language with 541 states and 403 symbols only contained 4151 useful entries, thus utilizing only 1.9% of the matrix.

An LR automaton can also be represented as a directed graph, where each vertex of the graph represents a state of the automaton and each edge, labeled with a symbol, represents transitions between states. (In this representation, one must be able to differentiate between lookahead states and regular states. In addition, special *reduce states* (called "lookahead states" in [6]) and *shift-reduce states* must also be used for reduce and read-reduce actions, respectively.) This representation has the advantage of compactness since it needs only be as large, proportionally, as the number of significant entries in the corresponding matrices. However, the computation of an action using this representation is slow since it may require that each out-edge connected to a state be explored.

From this discussion, one gathers that a desirable goal in generating LR parsing tables is to find a representation that performs as well as the matrix in terms of time-efficiency but whose space requirement is close to that of the graph representation.

A number of methods are known in the art for the compression of LR parsing tables. The compression can be achieved, for example, by the use of hashing, linear lists [32], row-displacement [32, 30, 23, 15], or graph-coloring [30].

Hashing is a standard technique for storing and searching large sparse tables, but it is seldom used for LR parsing applications because of its poor worst-case performance. It also consumes a large amount of space.

Substantial space savings result when the significant entries in each row of a matrix are stored in a linear list. The list, however, must be searched sequentially when a parse action is needed. Therefore, the time required to determine a parse action is not constant, but depends on the number of significant entries in the state in question. This method, discussed in [32], does save space but at the expense of time.

In the row-displacement method, the rows of a sparse matrix are *overlaid* on each other in a one dimensional table and an auxiliary table is used to retrieve the starting index of each row in the overlay table. Each entry in the overlay table contains a *check* field that is used to verify whether or not that entry corresponds to the original useful entry in the matrix. This method is advocated by Aho, Sethi and Ullman in [32] and by Ziegler in [15], and is discussed in detail in [30] and [23]. It is also used together with list searching in the YACC parser generator, as described in [32]. The row-displacement method does well with respect to time-efficiency, but its space utilization is not always optimal unless the matrix in question has a "harmonic decay" property [23] and a *first-fit decreasing* heuristic is used.

In the GCS (graph-coloring scheme) method proposed by [30], the rows of the original matrix are partitioned into classes of rows that do not have different significant entries in any column position. These rows are then merged to form a shorter matrix. Next, the columns of this new matrix are partitioned and merged, using the same method, to obtain a final reduced matrix. A vector *rowmap* indexable by the row indexes of the original matrix and a vector *columnmap* indexable by the column indexes of the original matrix are used to map the row and column indexes of the original matrix into the row and column indexes of the reduced matrix. In addition, when this method is used to compress the ACTION table of an LR parser, a boolean matrix *sigmap*, indexable by the indexes of the original matrix, is required to validate entries in the reduced matrix. If the boolean values in *sigmap* are contained in a single bit, this method yields relatively small parse tables where actions can be computed with a constant number of operations. However, since accessing a bit element of a matrix is usually a complex operation in most machines and *sigmap* must be accessed for each terminal action, the time-performance of this method may not be so good.

In the method presented here, the compaction of LR parsing tables is performed in three steps. In the first step, a number of transformations are applied on the original parsing tables to significantly reduce the number of rows and useful entries in them. (The resulting tables usually have the harmonic decay property.) In the second step, row-displacement is used to compact the resulting transformed matrices. The final step is a clean-up step where the actions in the compressed table are updated and all auxiliary tables are eliminated. This approach often yields perfectly compacted tables that require a fixed number of primitive operations on integers to compute an action.

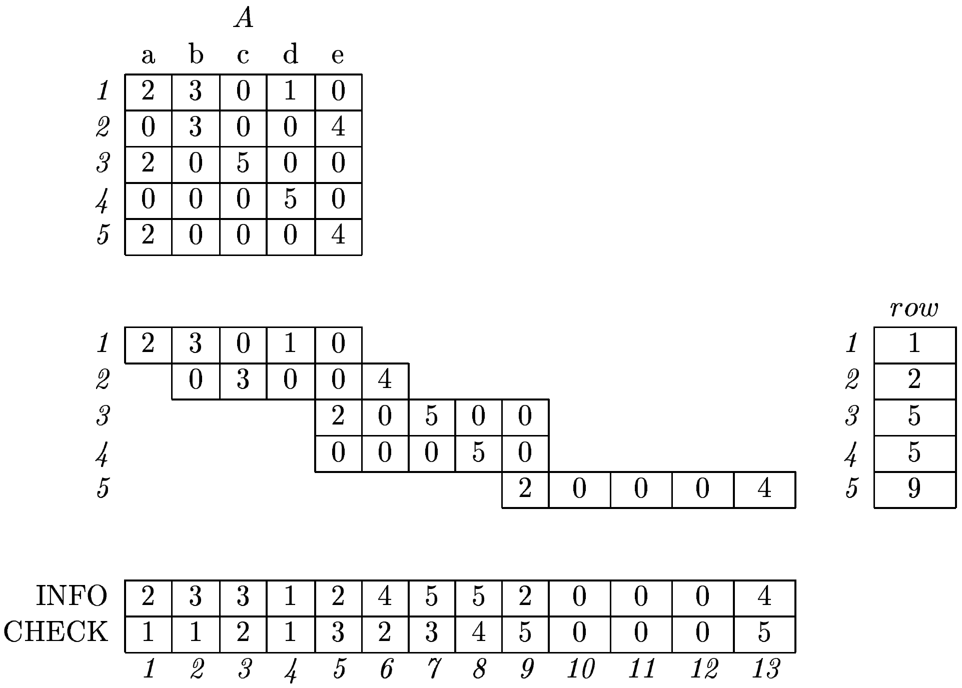
5.1.1 The Row-Displacement Scheme

Let A be a sparse matrix with m_1 rows, m_2 columns and n useful entries. Such a matrix A is usually stored in row-major order as a one-dimensional array A' containing $m_1 \times m_2$ elements. In this representation, each element $A(i, j)$ of the matrix corresponds to the element $A'((i-1) * m_1 + j)$ of the one-dimensional array. Let row be an auxiliary one-dimensional array of m_1 elements, each containing the offset of its corresponding row in the matrix; i.e., $row(i) = ((i-1) * m_1)$, $1 \leq i \leq m_1$. Using row, an element $A(i, j)$ corresponds to the element $A'(row(i) + j)$.

The row-displacement scheme is a method for compressing a sparse matrix A into two parallel one-dimensional arrays, CHECK and INFO, with fewer positions than A'. Given a pair (i, j), the CHECK array is used to test whether or not that pair is associated with a useful entry in A. If it is, the relevant information is retrieved from the INFO array.

The compaction is performed by overlapping the rows of A and placing their values in INFO in such a way that no two useful entries end up in the same position. The algorithm can be stated more formally as follows. For each row i in A, a sequential search is performed on the elements of INFO until a set of valid positions is found that can accommodate the useful entries of row i. Next, the ith entry of the row vector is initialized with the displacement of row i in INFO; each useful entry $A(i, j)$ is placed in its assigned location, INFO$(row(i) + j)$, and the corresponding CHECK$(row(i) + j)$ element is set to i to indicate that the element $row(i) + j$ of INFO is an element of row i in A. This algorithm is known as the "*first-fit*" method. (It is NP-complete to find a set of displacements that minimizes the size of the compressed vector [23] [16]).

Figure 5.1 depicts a matrix A and a possible compressed representation for that matrix. Non-useful entries in A contain the value 0. The columns of A are labeled with the sequence of letters [a,b,c,d,e] to avoid confusion. These indexes can easily be mapped into the integers [0..4] as the row-displacement scheme assumes that the indexes of the matrix are integers. To look up an element $A(i, j)$, one checks whether or not CHECK$(row(i) + j) = i$. If the test succeeds then INFO$(row(i) + j)$ is the relevant value of $A(i, j)$; otherwise, $A(i, j)$ is not a useful entry and 0 is the relevant value.



A

	a	b	c	d	e
1	2	3	0	1	0
2	0	3	0	0	4
3	2	0	5	0	0
4	0	0	0	5	0
5	2	0	0	0	4

1	2	3	0	1	0								
2		0	3	0	0	4							
3					2	0	5	0	0				
4					0	0	0	5	0				
5									2	0	0	0	4

	row
1	1
2	2
3	5
4	5
5	9

INFO	2	3	3	1	2	4	5	5	2	0	0	0	4
CHECK	1	1	2	1	3	2	3	4	5	0	0	0	5
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*

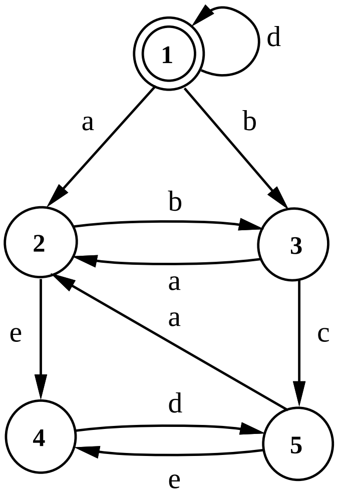
Fig. 5.1: A matrix A and its compressed representation

Note that if each row of a matrix has a unique displacement in its compressed representation then the column indexes of the matrix can be used as check values instead of the row indexes. This is particularly useful if $m_1 \ll m_2$ (say $m_1 < 256$) and can be stored in a smaller storage unit than m_2. Note also that the lower bound of the compressed vector does not have to be 1, but instead can be an arbitrarily chosen integer.

Assume that the matrix A of Figure 5.1 is a table representation of the transitions of a finite state automaton (Figure 5.2(a)). Assume that state 1 is the root state and as before, the edge labels [a,b,c,d,e] are mapped into the integers [0..4]. In such a case, as was stated earlier, the direct access capability of the matrix does not need to be preserved and the auxiliary displacement vector can be eliminated by replacing each useful entry in INFO by its displacement value. See Figure 5.2(b). Of course, in such a case the displacement of the starting state must still be stored in an auxiliary *Start* variable.

Ziegler suggested that the rows of a matrix to be compacted with the row-displacement scheme be sorted in decreasing order by the number of useful entries they contain before applying the compaction algorithm. This approach is known as the "*first-fit decreasing*" method. Tarjan and Yao [23] showed that this approach always produces a perfectly compacted vector "if the distribution of nonzeros among the rows are reasonably uniform". Their analysis was stated as follows:

> Our intuition is that rows with only a few nonzeros do not block too many possible displacement values for other rows; it is only the rows with many nonzeros that cause problems. To quantify this phenomenon, we define $n(l)$,



(a)

															Start
INFO	2	3	3	1	2	4	5	5	2	0	0	0	4		1
CHECK	1	1	2	1	3	2	3	4	5	0	0	0	5		
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*		

(b)

Fig. 5.2: A finite state automaton and its compressed representation

for $l \geq 0$, to be the total number of nonzeros in rows with more than l nonzeros. Our first theorem shows that if $n(l)/n$ decreases fast enough as l increases, then the first-fit decreasing method works well.

Theorem 5.1.1: Suppose the array A (to be compressed) has the following "harmonic decay" property:

$$\text{H: For any } l,\ n(l) \leq n/(l+1).$$

Then every row displacement $row(i)$ computed for A by the first-fit decreasing method satisfies $0 \leq row(i) \leq n$.

. . . If A has harmonic decay, at least half the nonzeros in A must be in rows with only a single nonzero. In addition, no row can have more than $\sqrt{n}$ nonzeros.

5.1.2 Transformations

Given an augmented grammar $G = (N, T, P, S)$ and its LALR(k) parser with varying-length lookahead strings and LR(0) reduce states removed, an entry in the ACTION matrix of such a parser can have one of six values. They are:

- *shift p*
- *reduce* $A \to \omega$
- *shift-reduce* $A \to \omega$
- *la-shift* p'
- *accept*
- *error*

where p is an LR(0) state of the parser, $A \to \omega \in P$ and p' is a lookahead state. The *error* parsing action is a constant value that is associated with each element of the ACTION matrix whose indexes correspond to a state-terminal pair on which no action is defined in the automaton. The parsing action *accept* is also a constant value. When an LR parser is constructed from an augmented grammar, *accept* is associated with a single entry in the matrix. During parsing, *accept* signals the successful completion of a parse.

An entry in the GOTO table can have one of three values. They are:

- *goto p*
- *goto-reduce* $A \to \omega$
- *don't care*

where p and $A \to \omega$ are defined as before. A non-useful entry in the GOTO table is referred to as a *don't care* entry instead of as an *error* entry because such an entry is never consulted during parsing.

5.1.2.1 GOTO Default Actions

Since "don't care" entries in the GOTO table are never accessed, a significant decrease in the number of useful entries in that table can be obtained by introducing default actions on nonterminals [32]. In this method, this is achieved as follows. Let GOTO_DEFAULT be a vector whose elements are indexed by the nonterminals. For each nonterminal A, scan the column of GOTO indexed by A and find the most frequent action, *act*, in that column. Remove all occurrences of *act* found in the A column and set GOTO_DEFAULT(A) = act. Looking at this transformation from the point of view of the directed graph representation, this is equivalent to removing all incoming edges labeled A that point to the state associated with the action *act*. (If *act* is a shift or lookahead-shift then all the incoming edges of its corresponding state are labeled A.) When actions are removed from the GOTO table, these entries and all the "don't care" entries are replaced by error entries. During parsing, if a state-nonterminal pair (p, A) yields *error*, GOTO_DEFAULT(A) is used.

5.1.2.2 Merging of Compatible States and Default Reductions

In chapter 3, the concept of default reduce actions was introduced as a way to optimize space in the ACTION matrix. In this method, the "default reduce actions" optimization is combined with the *merging of compatible states*. Together, these two optimizations result in a significant decrease in the space requirement of the ACTION matrix.

Definition 5.1.1: A state p is said to be compatible with another state q if and only if the following conditions are satisfied:

- p has the same terminal transitions as q; i.e., the rows of the ACTION matrix associated with p and q have the same shift, shift-reduce and lookahead-shift actions defined in their corresponding columns.
- The set of rules on which reduce actions are defined in q is the same as the set of rules on which reduce actions are defined in p.
- For each terminal a on which a reduce action is defined in both p and q, the reduce action is the same.

Two compatible states p and q can be merged into a single state r where the terminal transitions of r are the same as the terminal transitions of p (or q) and the reduce actions of r is the union of the reduce actions in p and the reduce actions in q. During parsing, if state r is entered in a context where p should have been entered and a reduce action that originated solely from q is executed, that action has the same effect as a default reduction. Similarly, if r is entered instead of q and a reduce action of p is executed in r, it can be viewed as a default reduction. Thus, unlike the method of [30], when states are merged based on the above criteria, the resulting merged states retains their error detecting capabilty and no external boolean check matrix is needed.

The set of states in a parser can be partitioned into a set of *compatible classes* based on the concept of compatible states described above. In other words, a compatible class of states is a subset of the set of states where any two states in the class are compatible. In general, finding an optimal partition of compatible classes which minimizes the number of subsets in the partition is NP-complete. However, the practical goal one tries to achieve in merging compatible states is to take advantage of the fact that most languages, especially programming languages, contain some phrases that are used in different contexts, but the actions induced by these phrases in the different contexts are compatible. For example, most procedural languages support arithmetic and relational expressions which are derived from a generic nonterminal expression and used in different statements such as:

$$\begin{array}{lcl} \mathsf{statement} & \rightarrow & \mathsf{variable} := \mathsf{expression}; \\ & | & \textbf{while}\ \mathsf{expression}\ \textbf{loop} \ldots \\ & | & \textbf{until}\ \mathsf{expression}\ \textbf{loop} \ldots \\ & | & \textbf{for}\ \mathsf{expression}\ \textbf{do} \ldots \\ & | & \textbf{if}\ \mathsf{expression}\ \textbf{then} \ldots \end{array}$$

Given a set of states of an LR parser whose kernel items are all of the form $[\alpha \cdot A\beta]$, where α and β are arbitrary strings, any two such states satisfy conditions 1 and 2 of the compatibility test. In the above example, the nonterminal expression appears in phrases that are all produced from the same source, statement, and no two of these phrases share the same prefix in that context. The LR parser for a grammar containing such rules will contain a state for each rule whose sole kernel item is the item derived from the right-hand side of the rule in question with the "dot" in front of expression. These states form a compatible class.

Experiments have shown that, in most cases, a set of states that meets conditions 1 and 2 of the compatibility test above is made up of states with kernel items having identical dot symbols. This is an important observation because it is condition 3 that renders the problem of finding the smallest partition of compatible classes NP-complete (see Appendix B for proof). To take advantage of this, a practical algorithm for merging states can be implemented in three steps based on the three conditions of the compatibility steps as follows:

step1: Construct a coarse partition based only on condition 1; i.e., states are grouped together if their terminal transitions are identical.

step2: Each partition class obtained from step 1 is broken down into smaller classes by applying condition 2.

step3: Finally, each class C obtained from step 2 whose states contain reduce actions by more than one rule is further fragmented by repeating the following heuristic process until C is empty:

> Remove an arbitrary state from C and use it to initialize a new singleton class C'. Next, each state p in C is considered, in turn. If p is compatible with all the states in C', it is removed from C and added to C'.

Once the states of an LR parser have been partitioned into compatible classes, each class of states is merged into a single merged state. The new merged states obtained by this process are used to form a new ACTION matrix. An extra column indexed by a special *default symbol* is appended to the front of ACTION. A default reduce action is computed for each merged row and that value is entered in the default column position of the row in question. The default reduce action of a given row, as described before, is the most frequently occurring reduce action in that row if there is any, otherwise it is the error action.

Now, since the terminal transition entries in the new ACTION matrix still refer to the original state numbers and having merged the rows of the ACTION matrix independently of the GOTO table, one must be able to relate each original state of the parser (still associated with a row of GOTO) with its merged counterpart in ACTION. This can be achieved by adding a new column to the GOTO table, each element of which points to its associated merged row in the new ACTION matrix. From now on, ACTION will denote the new matrix obtained after compatible states have been merged and default reductions computed.

5.1.3 Compaction of the Parsing Tables

Recall that in order to execute a reduce action by a given rule, the length of the right-hand side and the left-hand side nonterminal of the rule must be known. Therefore, to accommodate these actions, a vector, RHS_SIZE, indexable by rule numbers is initialized with the length of the right-hand side of each rule and a vector, LHS, also indexable by rule numbers is initialized with the left-hand side symbol of each rule.

LR parsers have two important properties which will be used as the basis of some important optimizations to improve both space utilization and the time performance of the compacted tables. They are reviewed here:

1. Since each time a parsing action is executed it determines the next state that will be entered, the direct access property of the matrix does not have to be preserved. In other words, since the parser can only enter a state p if a read transition places it in p or a read transition had previously placed p in the state stack and a reduce action re-exposed p on top of the stack by popping states corresponding to the right-hand side of a rule, only actions that are read transitions to state p need to be able to access p.

2. Another automaton, isomorphic to the original automaton, that recognizes the same language may be constructed by permuting and/or relabeling the rows and columns of the parsing tables, and changing the read transition entries accordingly. Viewed as a graph, permuting the rows constitutes a relabeling of the vertices (states) of the graph; and likewise, permuting the columns constitutes a relabeling of the edges of the graph or a renaming of the symbols.

5.1.3.1 The GOTO Matrix

Usually, the GOTO matrix contains many rows with no useful entries in them. After nonterminal actions are removed from GOTO by default, the usually sparse matrix ends up being even sparser. With the addition of the column that identifies the corresponding rows in ACTION as the first column in GOTO, many rows that previously had no action end up with a single action and the resulting matrix almost always has harmonic decay.

Before compacting the GOTO matrix, its columns are sorted in decreasing order by the number of useful entries they contain. The GOTO matrix is then compacted into two one-dimensional arrays, GOTO_CHECK and GOTO_INFO, with lower bound $|P| + 1$.

Since each element of the identification column contains a useful entry, this column will be the first column in GOTO after the columns have been sorted. It follows that each row of GOTO will have a unique displacement in its compressed representation and the column indexes (nonterminal symbols) of GOTO can be used as check symbols in GOTO_CHECK. Furthermore, since the application of default actions to GOTO usually removes all useful entries in some columns, the sorting places these empty columns in the upper range of the column indexes - minimizing the index values that may appear in GOTO_CHECK. Finally, observe that the elements of GOTO_CHECK that correspond to a state identification entry in GOTO_INFO can be used for any purpose since they are not associated with any parsing action. One only has to ensure that a value placed in these locations does not fall in the range of the column indexes of GOTO.

5.1.3.2 Encoding of the Parsing Actions

Since the *goto* actions in GOTO_INFO represent the transitions of a finite state automaton, each such state entry can be replaced by its corresponding row displacement. This mapping eliminates the need for a displacement vector. Furthermore, it allows a convenient encoding of the different kinds of values that may appear in the parsing tables. Let m be the highest displacement associated with a state, any integer r such that $1 \leq r \leq |P|$ represents a rule; any integer q such that $|P| < q \leq m$ can potentially represent a state; the *accept* action is represented by the constant $m + 1$ and the *error* action is represented by $m + 2$; an integer sr such that $error< sr \leq error+|P|$ represents a shift-reduce action; finally, an integer ls such that $error+|P| < ls$ potentially represents a lookahead-shift action.

With this encoding, one can easily identify what kind of action a given entry in a compressed parsing table represents. Reduce and goto-reduce actions are represented by the number r associated with the rule in question. Shift and goto actions are represented by the number q associated with the state in question. Shift-reduce actions are encoded as described above and the constant *error* must be subtracted from these values to obtain the rule in question. Similarly, the constant $error+|P|$ must be subtracted from the value of a lookahead-shift action to obtain the lookahead state in question.

After the GOTO table has been compressed, enough information is known to encode all parsing actions in both matrices except the lookahead-shift actions since these actions are transitions to lookahead states that are associated with rows in ACTION that have no counterpart in GOTO. The initialization of the state identification entries in GOTO_INFO must also be deferred until after ACTION has been compacted. Thus, a clean-up phase is required after the compaction of ACTION has been performed. This will be discussed later.

5.1.3.3 The Action Matrix

A typical LR parser usually contains many states (but less than half the total number of states) on which only a single terminal action is defined. The removal of many reduce actions by default further increases the number states with a single action as well as the sparseness of the ACTION matrix. If an optimized ACTION matrix is considered without its default reduction column, it usually has harmonic decay. Therefore, a good theoretical approach for compacting ACTION is to remove the default reduction column from it, store the default reduce action associated with each state in the unused entry in GOTO_CHECK associated with that state, and compress ACTION without the default reduction column. Note that when this approach is used, the rows of ACTION will not automatically have unique displacements. From a practical point of view, it may be beneficial to enforce this restriction because a typical grammar contains fewer than 255 terminals but a parser for such a grammar usually contains several hundred states. Therefore, if the column indexes of ACTION can be used as check values they can be accommodated in a single *8-bit byte*. On the other hand, the number of states may not only exceed 255 but, moreover, when states are replaced by their displacements, this further increases their range.

A second approach is to assume that each entry in the default reduction column is useful and sort the columns of ACTION in decreasing order by the number of useful entries in them. As in the case of the GOTO matrix, this guarantees that a column containing only useful entries will be the first column in ACTION and each row of ACTION will have a unique displacement in the compressed representation. ACTION is compressed into two one-dimensional arrays, ACTION_CHECK and ACTION_INFO, with lower bound 1.

If the input grammar used to generate an LR parser does not contain useless nonterminals (nonterminals that do not produce any terminal string), each state of the parser will have at least one terminal action. In that case, when the entries of the default reduction column are considered to be useful entries, each row of ACTION has at least two useful entries and thus, ACTION does not have harmonic decay. However, experiments have shown that when the columns of ACTION are sorted in decreasing order prior to applying the first-fit decreasing method, this rearrangement tends to produce good compaction. This second approach for compacting ACTION is the one that is used in this method. For practical reasons which will become clearer later, it turns out that in most cases, this approach saves space and, in any case, it produces tables that give a slightly faster time performance.

5.1.3.4 Compressed Tables Update

In this section, the final clean-up phase of the compaction procedure is described. Let d be the displacement (in ACTION_INFO) of a row associated with a lookahead state. Each entry in ACTION_INFO that is a transition into that lookahead state is replaced by the value *error*$+|P| + d$. Thus, when a lookahead-shift action is decoded, it is the displacement of the relevant lookahead state in ACTION_INFO that is obtained. Next, the displacement element in GOTO_INFO associated with each state is updated with the displacement of its corresponding ACTION row in ACTION_INFO. The encoding of all parsing actions in the compressed tables is now complete.

In order to avoid having to check for boundary conditions during parsing, one must be able to check whether or not a useful entry is defined for any state-symbol pair. This implies that the GOTO_CHECK vector must contain at least $m + |N|$ elements, where m is the highest displacement associated with a state. Similarly, the ACTION_CHECK vector must be extended to contain $m' + |T|$ elements, where m' is the highest displacement

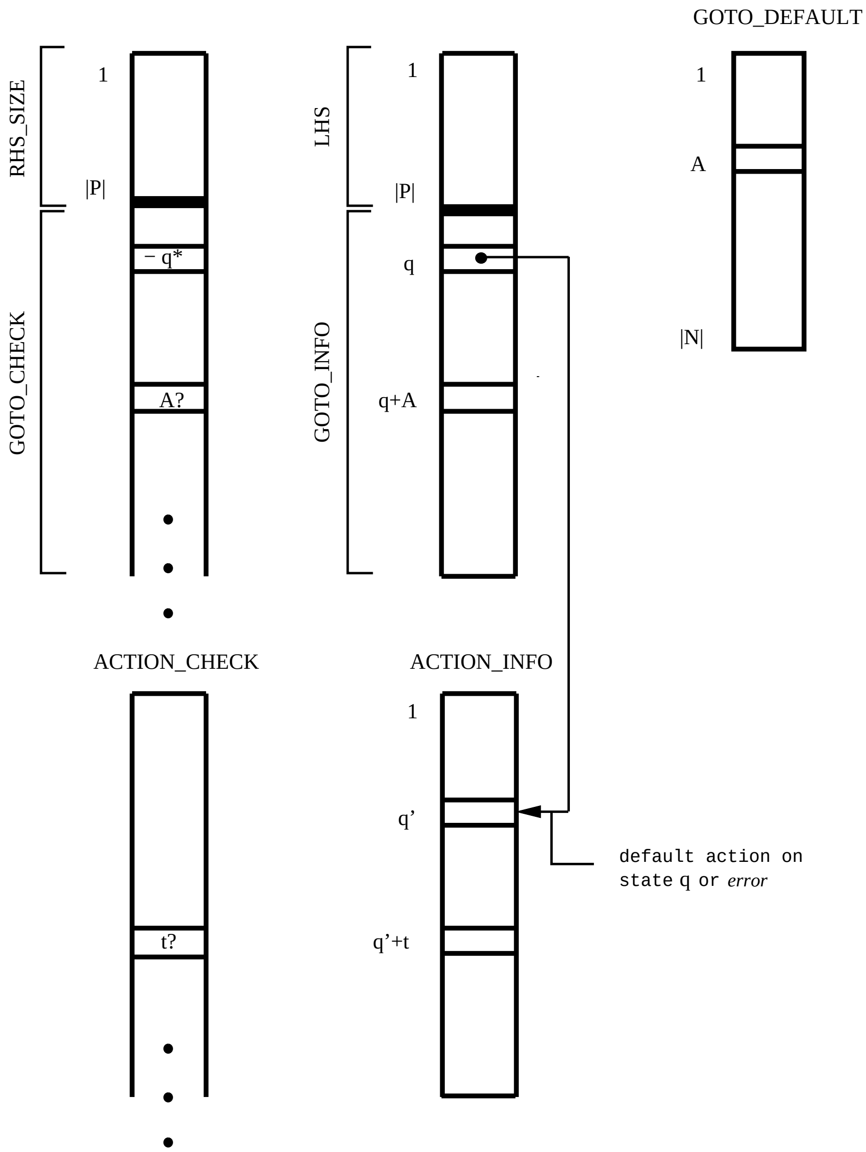


Fig. 5.3: Compressed parsing tables

```
-- Assume NUM_RULES = $|P|$ and LA_OFFSET = error + NUM_RULES.

function ACTION(q, $t_1..t_k$);
    q' := GOTO_INFO(q);
    if ACTION_CHECK(q' + $t_1$) = $t_1$ then
        act := ACTION_INFO(q' + $t_1$);
    else act := ACTION_INFO(q');
    end if;
    i := 1;
    while(act > LA_OFFSET) loop
        i := i + 1;
        q' := act - LA_OFFSET;
        if ACTION_CHECK(q' + $t_i$) = $t_i$ then
            act := ACTION_INFO(q' + $t_i$);
        else act := ACTION_INFO(q');
        end if;
    end loop;
    return act;
end ACTION;

function GOTO(q, A);
    if GOTO_CHECK(q + A) = A then
        return GOTO_INFO(q + A);
    else return GOTO_DEFAULT(A);
    end if;
end GOTO;
```

Fig. 5.4: Parsing functions

associated with an ACTION row.

Finally, since each element of GOTO_CHECK whose location corresponds to the location of a state identification entry in GOTO_INFO is still unused, the original row index (negated to differentiate it from check values) associated with the state in question can be stored at that location in GOTO_CHECK. This information is not needed for parsing (even though it might be useful for debugging purposes), but it provides a convenient remapping of each displacement into its original state number which is useful for storing the error recovery maps *t_symbols* and *nt_symbols*. The domain of these two maps is the set of states of the parser. This remapping allows the states to be represented by the row indexes of GOTO instead of the displacements in GOTO_INFO which fall in the larger range $|P| + 1..accept{-}1$.

Note that when this remapping of the states is used with the first approach for compacting ACTION discussed in the previous section, an additional DEFAULT vector, indexable by the states, must be used to store the default reduce action associated with each state. This arrangement not only requires extra space for the new vector but it also requires an additional indexing operation to compute a default reduce action. Experiments have shown that the number of useful entries in an optimized ACTION matrix is usually so small that the extra space lost by not obtaining a perfect compaction using the second approach is exceeded by the space required to store the DEFAULT vector. The computation of a terminal action with the second approach always requires the same number of operations.

5.1.3.5 Computing Parsing Actions on the Compacted Tables

Figure 5.3 shows a graphical representation of the compacted tables. ($q*$ denotes the row index of GOTO with displacement q.) Given such a representation the ACTION and GOTO functions of Figure 5.4 compute the action associated with a state-terminal or state-nonterminal pair, respectively. Note that if the tables were constructed from an LALR(1) parser, the code segment in lines 7 through 15 of the ACTION function can be omitted. The value returned by the ACTION function is either a reduce action if it is less than or equal to NUM_RULES, a shift action if it is greater than NUM_RULES but less than *accept*, or a shift-reduce action if it is greater than *error*. In the case of a shift-reduce action, *error* is subtracted from the value returned by ACTION to obtain the rule in question. The value returned by the GOTO function is either a goto-reduce action if it is less than or equal to NUM_RULES or a goto action, otherwise.

5.2 Error Recovery Tables

The error recovery method proposed in the previous chapter requires that some information be precomputed from the input grammar and its automaton. This information includes the scopes, discussed in section 4.3.3.1, the map *t_symbols* from each state q into a relevant subset of viable terminal error recovery candidates for q, the map *nt_symbols* from each state q into a relevant subset of viable nonterminal error recovery candidates for q and the *names* associated with each viable error recovery candidate.

$$\begin{array}{ll} E \to \cdot E + T & E \to \cdot T \\ T \to \cdot T * F & T \to \cdot F \\ F \to \cdot F \uparrow P & F \to \cdot P \\ P \to \cdot id & P \to \cdot (E) \end{array}$$

Fig. 5.5: Items in a state q_i

5.2.1 Optimization of Candidates

Consider the case of a secondary substitution in which a recovery goal A must be inserted into the input stream. In such a case, every nonterminal candidate in the recovery state q_i in question is a potential reduction goal. However, an implementation that checks all potential candidates for each error phrase would be prohibitively slow.

Two optimizations are applied to the set of nonterminal candidates in a given state to obtain, in most cases, a substantially reduced subset of *relevant reduction goals.*

In [27], the following concept is presented: a reduction goal A of error phrase $\beta|x$ in error configuration $\alpha\beta|xy$ is *important* if $\beta|x$ has no reduction goal B such that $B \to^+ A$. In this method, a more restricted concept of an important symbol is used. The new concept takes into consideration the full context of the error phrase:

Definition 5.2.1: A nonterminal A on which a transition is defined in a state q_i is said to be *important* if A does not appear in a single item of the form $B \to \cdot A$ in q_i.

For example, assume a recovery state q_i contains the set of items shown in Figure 5.5. By the definition of [27], the only important reduction goal in such a state is E, since T, F and P can be derived from E via a chain of unit productions. By the more restricted definition of this method, T and F would also be considered important symbols since they appear immediately to the right of the dot in more than one item. To understand the importance of T and F, assume that the rules from which the items of Figure 5.5 are derived are all the productions of a grammar and consider the following erroneous input strings:

```
( ) ) ( * id + id
( ) ) ( ↑ id + id
```

If E is the only important symbol considered, then the best secondary repair that is achievable is the replacement of "`( ) ) ( * id`" by E in the first sentence and "`( ) ) ( ↑ id`" by E in the second sentence. However, it is clear from the grammar that replacing "`( ) ) (`" by T in the first sentence and by F in the second sentence would be preferable.

One further notices that using F as a reduction goal in the first sentence would have worked just as well, since after a transition on F, with the symbol "*" as lookahead, a reduction by the rule "$T \to F$" would be applied. Similarly, P could have been used as a suitable reduction goal in both sentences. This leads to the following concept, on which the second optimization is based:

Definition 5.2.2: A nonterminal element C of a set of nonterminal candidates S in an LR state q is said to be *relevant* with respect to S if there does not exist a nonterminal D, such that $D \in S$, $D \neq C$, and D can be successfully substituted for C as a reduction goal for any error phrase with q as the recovery state.

Given a set S of nonterminal candidates for a given state, the objective is to find the largest subset $S' \subseteq S$ such that S' contains only relevant reduction goals.

Lemma 5.2.1: Let $S = \{B_1, \ldots, B_k\}$; for $1 \leq i \leq k$, $B_i \in S$ is relevant iff $\nexists B_j$, $j \neq i$, such that $B_i \Rightarrow_{rm}^{+} B_j$.

The proof follows directly from the definition of an LR parser. If a nonterminal B can be substituted for an error phrase, then the recovery symbol t in question must be a valid lookahead symbol for any rule derivable from B. In particular, if $B_i \Rightarrow_{rm}^{+} B_j$ and B_j is substituted for an error phase where B_i is known to be a valid reduction goal, the recovery symbol will cause $B_j$to be reduced to B_i.

For each state q in an LR automaton, the set *nt_symbols*(q) is obtained as follows. Starting with the set of nonterminal symbols on which an action is defined in q, remove all unimportant symbols from that set, and reduce the resulting set further by removing all irrelevant reduction goals from it. For example, consider the state q_i of Figure 5.5. State q_i contains nonterminal transitions on the symbols E, T, F and P. The only unimportant symbol in that set is P. After P is removed, the irrelevant symbols E and T are removed from the subset $\{E, T, F\}$ leaving F as the only relevant reduction goal in q_i.

The notion of an important symbol can also be extended to terminal candidates in the *t_symbols* sets. Once again, consider the state q_i of Figure 5.5. This state contains a single terminal action on the symbol *id*, but, since *id* appears only in the item $P \rightarrow \cdot id$, it is not an important candidate in q_i. The removal of unimportant terminals improves the time performance of the primary recovery and saves space. However, it may suppress some opportunities for merging and misspelling corrections.

In section 5.2.3, an algorithm is presented that can be used to further reduce the space used by *t_symbols* and *nt_symbols*.

5.2.2 Optimization of Names

In order for the error recovery to issue meaningful diagnoses, the names of the symbols used in the grammar should be saved. In fact, it is very helpful to allow the user to map terse symbol names used in his grammar into more descriptive names. For example, a nonterminal “relop” might be mapped into the string “relational operator”; the end-of-file terminal in an Ada grammar might be mapped into the string “End of Ada source”, etc.

Depending on the nature of an input grammar, some nonterminals may never be used in issuing a diagnosis. Therefore, the names associated with these symbols can be removed. Two kinds of nonterminals fall in this category: nullable nonterminals and nonterminals that are always subsumed by another nonterminal. The processing of nullable nonterminals in issuing diagnostic messages was discussed in section 4.4.3. A nonterminal B is said to be always subsumed by another nonterminal if whenever B appears in the right-hand side of a rule, the rule in question is a single-production of the form $A \rightarrow B$, for some nonterminal A. In such a case, B (actually, the name associated with B) will never appear in a diagnostic message since the left-hand side symbol A in question would be a higher symbol (see section 4.4.3) than B.

5.2.3 Optimization of Finite Subsets of a Small Universe

Observe that most of the information generated for error recovery consists of finite subsets of a small universe. If the only operation performed on these subsets is iteration, a sequential representation of these sets will yield a good time-performance. For example, *t_symbols* and *nt_symbols* are mappings from states into subsets of terminals and nonterminals, respectively. During error recovery, the only operation performed on these sets of candidates is iteration.

In practice, many of the subsets involved in these applications are identical and can share the same space. In addition, the resulting collection of unique subsets can be further optimized with a new algorithm, called Optimal Partition [39], that takes advantage of the fact that some subsets in that collection are proper subsets of others. Optimal Partition is used to optimize the subsets in the range of *t_symbols* and *nt_symbols* as well as the sets of states associated with the scopes. The operation performed on the set of states associated with a scope is a lookup. However, such a set is usually so small that, in practice, a look up on a sequential representation of it is acceptable.

5.2.3.1 The General Problem

Let Σ be an alphabet and let C be a collection of unique subsets of Σ:

$$C = \{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}.$$

Find a sequential representation of C that minimizes the storage space required such that the elements of each Σ_i, $1 \leq i \leq n$, occur in consecutive locations.

A set C is said to have the *consecutive retrieval property* with respect to a universe Σ if there exists an organization of Σ (without duplication of any symbol) such that for every $\Sigma_i \in C$ all relevant symbols of Σ_i can be stored in consecutive storage locations. A polynomial time algorithm for finding such an arrangement, if it exists, was presented by Fulkerson and Gross [2]. However, in most practical cases, a pair (Σ, C) does not have the consecutive retrieval property. Hence, duplication of symbols must be allowed so that pertinent symbols corresponding to any subset in C are always stored consecutively. This problem can be stated more formally as follows:

> Is there a string $w \in \Sigma^*$ with $|w| \leq K$ such that for each i, the elements of Σ_i occur in a consecutive block of $|\Sigma_i|$ symbols of w?

This problem known as Consecutive Sets(CS) was posed by Kou in [1] and proven to be NP-complete. Now, consider the problem of finding a string w, of minimum length, required to store the subsets of C such that the elements of each subset occur in a consecutive block of symbols of w and if any two subsets Σ_i and Σ_j share common elements then one is a proper subset of the other. This problem can be stated more formally as follows:

> Is there a partition P of C such that for a given integer K:
>
> - $\forall p_k \in P$, $\forall (x, y) \in p_k^2$, either $x \subset y$ or $y \subset x$.
> - Let $\hat{\Sigma}_k$ represent the largest subset in $p_k \in P$, then $\sum_{p_k \in P} |\hat{\Sigma}_k| \leq K$

This problem is the Optimal Partition problem, OP for short. In Appendix C, OP is proved to be solvable in polynomial-time by reducing it to the maximum weighted bipartite matching problem [39]. In the remainder of this section, a algorithm based on OP is used to optimize the space required to store a set of sets C sequentially, followed by the description of a time-efficient heuristic for OP which, in most cases, finds a partition that is very close to being optimal.

5.2.3.2 Application of OP

With OP, one can take advantage of the fact that there is no overlap of subsets in the elements of the partition P to construct a string w such that each subset can be identified with only one index indicating its starting point. Assuming # is a special marker symbol, $\# \notin \Sigma$, one proceeds as follows:

```
step0: initialize $w$ to the *empty string.*
step1: **for** each $p_k \in P$ **do** steps 1.1-1.3
  step1.1: construct $[\Sigma_{k_1}, \Sigma_{k_2}, ..., \Sigma_{k_m}, \Sigma_{k_{m+1}}]$, an ordered list of the elements
           of $p_k$, where $|p_k| = m$, $\hat{\Sigma}_k = \Sigma_{k_1}$, $\Sigma_{k_{m+1}} = \emptyset$, and $\Sigma_{k_i} \supset \Sigma_{k_{i+1}}$,
           $1 \leq i \leq m$. It is known that such a linear ordering of the elements of
           $p_k$ is possible by the definition of OP.
  step1.2: **for** i **in** $1..m$, append to $w$ all symbols that are in $\Sigma_{k_i}$ but not in
           $\Sigma_{k_{i+1}}$.
  step1.3: append to $w$ the symbol #.
```

5.2.3.3 OP Heuristic

Even though OP can be computed by a polynomial-time algorithm, the algorithm in question is not very efficient and in certain environments may prove to be impractical. A simple heuristic for OP will be described below. In most cases, this approach works very well and can be computed in $O(n^2)$ time.

A set $\hat{\Sigma}_k$ that is not included in any other set in a given partition is called a *base* set. The following heuristic is essentially a greedy algorithm that constructs the classes of the partition one at a time and tries to minimize the length of the final string by always including the largest subset it can find. The partition P is constructed as a set of ordered lists of subsets, where each such list make up a class of the partition and the order of the elements in the list is exactly the proper linear ordering for the subsets:

```
step 0: $P = \emptyset$;
step 1: *Construct a partition* $P' = \{p_i \neq \emptyset \ : \ i \in [1..|\Sigma|]\}$ *of* $C$, *where:*

            $p_i = \{\Sigma_k : \Sigma_k \in C \text{ and } |\Sigma_k| = i\}$

step 2: *do* steps 2.1 *and* 2.2
  step 2.1: *Let* $p_k$ *be the class in* $P'$ *with the largest subsets. Let* $x$ *be an arbitrary*
            *subset of* $p_k$, *remove* $x$ *from* $p_k$ *and initialize list* $:= [x]$;
```

```
    step 2.2: **if** $p_k = \emptyset$ **then** *remove* $p_k$ *from* $P'$; **end if**;
step 3: **for** i **in** $[k-1, k-2..1]$ | $p_i \in P'$ **loop**
            **if** $\exists\ y \in p_i \mid y \subset x$ **then**
                *remove* $y$ *from* $p_i$
                $list := list + [y]$;
                $x := y$
                **if** $p_i = \emptyset$ **then** *remove* $p_i$ *from* $P'$; **end if**;
            **end if**;
        **end loop**;
step 4: $P = P \cup \{list\}$;
step 5: **if** $P' \neq \emptyset$ **then goto** step 2; **else stop**; **end if**;
```

Using a bucket sort, step 1 can be computed in $O(|\Sigma| + n)$ time. For steps 2-5, the worst case occurs when no subset is a superset of another and each p_i contains exactly one element. In that case, each subset is removed, in turn, from $p_i \in P'$ and tested against the element in each p_j, $1 \leq j < i$. The total number of subset checks in that case is $n(n-1)/2$, which gives us a total running time of $O(n^2)$. Experimental results using both OP and this heuristic algorithm are presented in Figure C.1 in section C.3.

5.3 Remarks

When parsing tables are compressed with this method, each parsing action can be computed with a fixed number of primitive operations on integers and arrays of integers. For an LALR(1) parser, the computation of a terminal action costs exactly 3 indexing operations, 1 addition and 1 comparison. For an LALR(k) parser, in addition to the initial LALR(1) action, $k-1$ extra lookahead transitions may be required. Each lookahead transition costs 2 additions, 1 subtraction, 2 comparisons and 2 indexing operations. The computation of a nonterminal action always costs 2 indexing operations, 1 addition and 1 comparison.

Figure D.1 of Appendix D shows the space requirement of the parsing and error recovery tables of 6 programming language grammars.

CONCLUSION

A new method has been presented for constructing practical and efficient LALR(k) parsers with automatic error recovery. Significant contributions and improvements over the current state-of-the-art were made in three areas: parser generation, error recovery, table optimization and table compaction.

In the parser generator described in this thesis, only the minimum amount of lookahead information necessary (up to k levels) to disambiguate inconsistent states in the LR(0) automaton is computed. Consequently, this parser generator is efficient since, in practice, few states require more than one lookahead. Furthermore, the generated parsers are efficient, because at run-time the parser consults extra lookahead symbols only when necessary.

The LR(k) error diagnosis and recovery method presented here is completely language- and machine-independent and more efficient than other known methods. It features the following innovations and techniques:

- a new deferred driver that always detects an error at the earliest possible point;
- a generalized simple recovery method that uses both terminal and nonterminal symbols;
- a phrase-level recovery that is an efficient (and completely automatic) generalization of the error production method;
- a new, completely automatic method for scope recovery.

A number of innovative space optimization ideas are introduced in this thesis. The number of useful entries in the parsing tables and the error recovery maps are reduced and space is shared among similar sets. Following optimization, the parsing tables and the error recovery maps are compacted separately. This allows complete and accurate information to be retained. The resulting compacted parsing tables are the most time- and space-efficient known to date.

ACKNOWLEDGEMENTS

There are many people without whose help I would not have been able to reach this goal. I would like to give a very special thank you to Fran Allen for her encouragement, support and for giving me the opportunity to go back to school and finish my degree. I thank the IBM Corporation for its financial support. I give a special thanks to my advisor, Edmond Schonberg, for his guidance and insightful suggestions and to Gerry Fisher, my M.S. advisor, who got me interested in this field and with whom I developed the table compression algorithm. I thank the members of the PTRAN group: Mike Burke, Paul Carini, Ron Cytron, Jong Deok Choi, Jeanne Ferrante, Sue Flynn, Emily Plachy, Edith Schonberg and David Shields. Mike and I have been friends for many years. I am particularly thankful to him for helping me with my writing and for many useful technical discussions on error recovery and parsing in general. The error recovery method described in this thesis was inspired by previous work done by Mike and Gerry. Ron and I joined IBM around the same time and we have been friends ever since. Ron has always been very supportive and helpful. Jeanne reviewed some of my papers and made some helpful comments.

Many people have contributed some useful ideas to this work: Laurent Pautet was a co-author of the Optimal Partition solution; Franco Gasperoni proved the NP-completeness of the string compatibility problem; Fritz Henglein helped me on many occasions with some theoretical problems; Matthew Smosna made many worthwhile suggestions which found their way into this work; Yvonne Lesesne translated the original PL/I code to C; Kirk Snyder helped me uncover some bugs and offered many valuable suggestions. Some ideas presented in this work were developed as a result of requests made from the LPG user community at IBM - in particular, Kile Reeves and Ira Sheftman made some helpful suggestions.

Many thanks to my family, especially my mother. Special thanks to Andrea Erstling who has been very supportive and helped me edit this work. Thank you to all my friends, especially Elysée, Fritz, and Selena. I am very grateful to Professor Malcolm Brown of Brooklyn College, Mr Sherman Lee, my undergraduate advisor, and Professor Harry Hamilton of the State University of New York at Albany for their positive influence during my undergraduate years.

This work is dedicated to my aunt, Ms. Jeanne Leonard.

APPENDIX

A. EXPERIMENTAL RESULTS WITH THE LALR ALGORITHMS

In this appendix, experimental results obtained from six grammars of different sizes and complexity are presented. The table of Figure A.1 summarizes the characteristics of each of the grammars used. The Pascal1 and Pascal2 grammars are LALR(2) and the others are LALR(1). Pascal2 is the grammar of Appendix F. Ada is the grammar of [29]. Note that for the Pascal1 and Pascal2 grammars, under the "states" column two numbers are reported. The first is the number of LR(0) states and the second is the number of lookahead states. The automaton for these grammars also contains a lookahead-shift action for each lookahead state.

Grammar	$\|T\|$	$\|N\|$	$\|P\|$	#items	#states	#shifts	#shift-reduces	#gotos	#goto-reduces
Pascal	63	110	213	626	193	396	329	336	574
Pascal1	63	111	215	625	189/1	393	324	332	569
Pascal2	63	111	215	627	191/5	393	326	334	569
C	85	98	266	786	206	349	987	975	322
Ada	96	304	523	1527	516	593	700	884	1779
Sedl	123	362	746	2398	824	1987	5755	2250	4360

Fig. A.1: Grammar information

The first three columns in the table of Figure A.1 indicate the number of terminals, nonterminals and productions, respectively, in each grammar; the $\#items$ column indicates the total number of items in the grammar, i.e., let rhs_i represent the list of symbols on the right-hand side of production i, then $\#items = \sum_{i=1}^{|P|} |rhs_i| + 1$; the $\#states$ column indicates the number of states in the LR(0) automaton; the $\#shifts$ and $\#gotos$ columns indicate, respectively, the number of shift and goto transitions in the LR(0) automaton. Figure D.1 gives more information about these grammars and their automata.

The main goal of the experiments was to compare the time performance of each method in terms of the number of union operations it required to compute the LALR(1) lookahead sets for each grammar. Note that lookahead sets were computed only when necessary to resolve conflicts. In particular, lookahead sets associated with LR(0) reduce states were not computed. The KM, DP and PCC method were implemented. It was not necessary to implement the BL method since its time performance, in terms of number of union operations performed, is identical to the DP method.

The KM algorithm was implemented as described in section 2.2.1. The number of union operations required is reported under the $\#unions$ column. When the improvement suggested in section 2.3.1 is added, the number of union operations required is calculated as the number of visits ($\#visits$) made to TRANS. Recall that in such a case a preliminary pass is required to compute the DR sets for each state. The total number of elements in these sets is equal to *#shifts*. The data from the KM experiments is shown in figure A.2.

KM	*#visits*	*#unions*
Pascal	1176	3382
C	13651	36829
Ada	5861	12038
Sedl	30770	67479

Fig. A.2: Results of KM experiments

The table of figure A.3 shows the number of union operations required to compute the necessary FIRST sets. The $N - unions$ column indicates the number of union operations required to construct the FIRST sets for nonterminals. The $S - unions$ column indicates the additional operations required to compute the FIRST sets for suffixes that follow a nonterminal in a production of P. The cost of computing these maps was incurred by the DP and PCC algorithms.

Grammar	$N - unions$	$S - unions$
Pascal	137	10
Pascal1	137	11
Pascal2	137	13
C	166	0
Ada	372	5
Sedl	508	10

Fig. A.3: Unions required for FIRST maps

DP/Charles	#READ	#FOLLOW	#LA	*Total*
Pascal	93	769	384	1393
Pascal1	90	764	381	1383
Pascal2	91	764	415	1420
C	106	1684	1212	3168
Ada	132	1868	750	3127
Sedl	281	7083	3154	11036

Fig. A.4: Results of DP experiments

The DP method was implemented with the improvements suggested in section 2.3.2. The data from the DP experiments is shown in the table of figure A.4. The #READ column indicates the number of union operations required to construct the READ sets from FIRST sets. The #FOLLOW column indicates the number of union operations required to construct the FOLLOW sets from READ sets. The #LA column indicates the number of union operations required to construct the LA sets from FOLLOW sets. The *Total* column includes the union operations incurred by the construction of the FIRST sets.

The data from the PCC experiments is shown in the table of figure A.5. The #PATH column indicates the number of union operations required to construct the PATH sets. The #FOLLOW and #LA columns indicate the number of union operations required to construct the FOLLOW and LA sets, respectively. The *Total* column includes the union operations incurred by the construction of the FIRST sets.

PCC	#PATH	#FOLLOW	#LA	*Total*
Pascal	853	759	805	2564
C	1888	2114	2317	6485
Ada	3978	1824	1354	7533
Sedl	6886	7926	6154	21484

Fig. A.5: Results of PCC experiments

Finally, the table of figure A.6 shows how the different methods performed. Each entry in the table indicates the total number of union operations that were required by the particular method that labels the column and the grammar that labels the row.

Grammar	KM	DP/Ch	PCC
Pascal	1176	1393	2564
Pascal1		1383	
Pascal2		1420	
C	13651	3168	6485
Ada	5861	3127	7533
Sedl	30770	11036	21484

Fig. A.6: Comparison of the methods

B. STRING COMPATIBILITY

Let Σ be an alphabet containing a special blank character, denoted $\flat$. Consider a set S of n strings of Σ characters, each of length m:

$$S := \{s_1, s_2, \ldots, s_n\}.$$

A string $s_i \in S$ is said to be compatible with another string $s_j \in S$ if and only if for $1 \leq k \leq m$, either $s_i(k) = s_j(k)$ or $s_i(k) = \flat$ or $s_j(k) = \flat$. This compatibility relation is reflexive and symmetric, but not transitive.

The String Compatibility problem is: given some positive integer K, can S be partitioned into $k \leq K$ *compatible classes*. A *compatible class* of S is a subset of S whose elements are pair-wise compatible. The String Compatibility problem is NP-complete. This will be shown by reducing *clique cover* [8] (also known as *Partition into Cliques* [22]) to String Compatibility [43]. Recall the definition of Partition into Cliques:

> *Instance:* Graph $G = (V, E)$, positive integer $K \leq |V|$.
>
> *Question:* Can the vertices of the graph G be partitioned into $k \leq K$ disjoint sets $V_1, \ldots, V_k$ such that for $1 \leq i \leq k$, the subgraph induced by v_i is a complete graph (also called a clique).

Reduction

Given an instance of "Partition into Cliques", construct a *negative* graph $G' = (V, E')$, where $E' = V \times V - E$. Let each $e' \in E'$ be represented by an integer in the range $1..|E'|$ and let $\Sigma = V \cup \{\flat\}$. The set S is constructed by mapping each vertex v_i into a string s_i of length $|E'|$ and adding s_i to S. The mapping is done as follows. If $e' = (v_i, v_j) \in E'$ then $s_i(e') = v_j$; otherwise $s_i(e') = \flat$. This transformation can clearly be computed in polynomial-time.

Claim: a subset of the strings in S form a compatible class if and only if the corresponding vertices in G form a clique.

If: Let C be a subset of S that forms a compatible class, then for each pair of strings s_i and s_j in C, $(v_i, v_j) \notin E'$. Therefore, $(v_i, v_j) \in E$. It follows that the vertices that correspond to the strings in C induce a clique in G.

Only if: Conversely, given a clique in G and an edge (v_i, v_j) in that clique, the corresponding strings s_i and s_j are compatible. In fact, if s_i is not compatible with s_j then there exists an edge $e' \in E'$ such that $s_i(e') = v_j$ and $s_j(e') = v_i$. But, this implies that $(v_i, v_j) \in E'$ which is a contradiction.

Thus, G can be partitioned into k cliques if and only if S can be partitioned into k compatible classes. □

C. OPTIMAL PARTITION

C.1 Reduction to Maximum Weighted Bipartite Matching Problem

Given an alphabet Σ and a collection $C = \{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}$, where for each $\Sigma_i \in C$ and $\Sigma_i \subseteq \Sigma$, an effective method is presented to construct a bipartite graph $G = (V, V', E)$ with a weight function W defined on E such that a maximum weighted matching obtained from G defines an Optimal Partition for C with the elements of each class in the partition linearly ordered.

C.1.1 Bipartite Graph

The bipartite graph $G = (V, V', E)$ is constructed fron the collection C as follows. Firstly, each subset Σ_i is associated with a unique node $v_i \in V$ and a unique node $v'_i \in V'$. Next, two bijective functions f and f' are used to make the respective associations:

$$f : C \to V$$

$$f' : C \to V'$$

$$V = \{f(\Sigma_i) : \Sigma_i \in C\}$$

$$V' = \{f'(\Sigma_i) : \Sigma_i \in C\}$$

The node yielded by $f(\Sigma_i)$ is denoted v_i and the node yielded by $f'(\Sigma_i)$ is denoted v'_i. Two nodes $v_i \in V$ and $v'_i \in V'$ corresponding to the same subset Σ_i are said to be *twins.*

Let the operator $\subset$ ($\supset$) denote proper inclusion, i.e., $A \subset B$ ($A \supset B$) means that A is a subset (superset) of B, but $A \neq B$. A binary relation $R \subseteq C \times C$ is defined as follows:

$$R = \{(\Sigma_i, \Sigma_j) \mid \Sigma_i \supset \Sigma_j\}$$

R is transitive, non-reflexive, and non-symmetric. The set of edges $E \subseteq V \times V'$ in G is constructed as follows:

$$E = \{(v_i, v'_j) : v_i \in V, v'_j \in V' \mid (\Sigma_i, \Sigma_j) \in R\}$$

Given an edge $(v_i, v'_j) \in E$, v'_j is said to be a *successor* of v_i and v_i is a *predecessor* of v'_j. Let $M \subseteq E$ be a bipartite matching, not necessarily maximal, on G then lemmas 1 and 2 follow:

Lemma C.1.1: Each $v_i \in V$ is connected to at most one successor in M

Lemma C.1.2: Each $v'_j \in V'$ is connected to at most one predecessor in M

Proof: definition of a bipartite matching.

A node $v_i \in V$ having no successor is called a *terminal* node. A node $v_i \in V$ whose twin has no predecessor is called a *base* node. Let β be the set of base nodes induced by M and let γ be the set of terminal nodes induced by M,

Lemma C.1.3: $\beta \neq \emptyset$

Proof: Assume that $\beta = \emptyset$. Then every node in V is matched to some node in V'. This implies that for each subset $\Sigma_j \in C$, $\exists\ \Sigma_i$ and Σ_k in C such that $\Sigma_i \subset \Sigma_j \subset \Sigma_k$; but, this implies that the relation R is circular, which is a contradiction.

Corollary C.1.1: $\gamma \neq \emptyset$ and $|\gamma| = |\beta|$

Proof: By the same argument used for lemma C.1.3 above, $\gamma \neq \emptyset$. By definition of bipartite matching, it follows that the number of unmatched nodes in V must be equal to the number of unmatched nodes in V'.

Consider an arbitrary matching M and the sets β and γ induced by it. For every $v_i \in \beta$, let $T(v_i)$ be the set of nodes v_k such that there is a sequence $v_{k_1}, v_{k_2}, \ldots, v_{k_m}$ where $v_i = v_{k_1}$, $v_{k_m} \in \gamma$, and $(v_{k_i}, v'_{k_{i+1}}) \in M, 1 \leq i \leq m-1$. A function τ is defined on each element $\Sigma_i \in C$ where $v_i \in \beta$ as follows:

$$\tau(\Sigma_i) = \{f^{-1}(v_k) : v_k \in T(v_i)\}$$

A subset Σ_i corresponding to a base node v_i is referred to as a *base subset* and Σ_i is said to define the set $\tau(\Sigma_i)$.

Lemma C.1.4: Let $P = \{\tau(\Sigma_i) \mid \Sigma_i = f^{-1}(v_i) \text{ where } v_i \in \beta\}$ then P is a partition of C and the subsets contained in each element $p \in P$ can be linearly ordered by the relation R.

1. Given an arbitrary subset $\Sigma_k \in C$, it is included in one of the (*set*) elements of P.

   Proof: If Σ_k is a base subset then it is clearly included in $\tau(\Sigma_k) \in P$. If Σ_k is not a base subset then its corresponding node v'_k has a predecessor in M, say v_j, corresponding to a subset Σ_j. Since R is transitive and non-circular then so is M. Therefore, there exists a base subset Σ_i such that $(\Sigma_i, \Sigma_k) \in M^+$. Hence, Σ_k is contained in an element of P.

2. Given two base subsets Σ_i and Σ_k, if $i \neq k$ then $\tau(\Sigma_i) \cap \tau(\Sigma_k) = \emptyset$.

   Proof: Lemma C.1.1 and C.1.2 and definition of τ.

3. The total order of the elements of $\tau(\Sigma_i)$ can be determined from the ordering of the corresponding sequence of $T(v_i)$.

(1), (2) and (3) above imply that P is a partition of C and the subsets contained in each $p \in P$ can be linearly ordered according to R.

Lemma C.1.5: Let P be a partition of C such that the elements of each $p \in P$ can be linearly ordered, then P corresponds to a matching in G.

Proof: Consider each ordered sequence $p = [\Sigma_{k_1}, \Sigma_{k_2}, \ldots, \Sigma_{k_m}]$ in turn. Include the set of edges $\{(v_{k_i}, v'_{k_{i+1}}) \ : \ 1 \leq i \leq m-1\}$ in the matching.

Theorem C.1.1: P is a partition of C whose elements can be linearly ordered based on the relation R if and only if it is induced by a matching on the corresponding graph G.

The proof follows from lemmas C.1.4 and C.1.5.

C.1.2 Weight Function

A weight function W is defined on edges of G as follows:

$$W(v_i, v'_j) = |\Sigma_j|$$

Applying a maximum weighted bipartite matching algorithm to G yields a matching M that induces a set of base nodes: β, such that the total weight of M is:

$$\sum_{(v_i, v'_j) \in M} |\Sigma_j| = \sum_{v_j \notin \beta} |\Sigma_j|$$

Let σ be the total sum of the lengths of the elements of C:

$$\sigma = \sum_{i=1}^{|C|} |\Sigma_i|$$

Let κ be the sum of the lengths of the base subsets. κ can be computed given σ and the weight of the matching M, as follows:

$$\kappa = \sigma - \sum_{v_j \notin \beta} |\Sigma_j|$$

Recall that the elements of the base subsets are concatenated to construct the string w. Therefore, since σ is constant and the sum of the weights of the *non-base* subsets is maximized by the maximum weighted bipartite matching algorithm, it follows that κ is minimized. Hence, the final result which is restated as follows:

Theorem C.1.2: $min \ \kappa \iff min \ |w|$

The proof follows from Theorem 1 and maximum weighted bipartite matching.

C.2 Complexity

Assume that one can test whether or not a given set is a proper subset of another set in unit time. This is a reasonable assumption if the length of Σ is not too large and bit-strings are used to represent the subsets. With that assumption, the construction of the graph G from the collection C requires $O(n^2)$ operations.

Tarjan in [7] showed that the maximum weighted bipartite matching problem can be reduced to the max-flow/min-cost problem with no negative cost cycles. He also presented an algorithm for finding a minimum cost maximum flow f^* in a network with n nodes and m edges in $O(m|f^*|\log_{(2+m/n)} n)$ time. In this case, the cost of constructing the graph is dominated by the max-flow/min-cost algorithm and $|f^*| = n/2$. Hence, the total running time is:

$$O(nm \log_{(2+m/n)} n) \quad \Box$$

C.3 Experimental Results with OP

Experimental results with the *t_symbols* maps of the six programming language grammars mentioned in Appendix A are presented in figure C.1.

Grammar	$\lvert\Sigma\rvert$	*t_symbols*	merged	Heuristic	OP
Pascal	80	1051/193	425/73	173/29	172/28
Pascal1	63	1036/189	423/72	172/29	171/28
Pascal2	63	1040/191	425/73	172/29	171/28
C	85	2598/206	1153/88	478/32	451/30
Ada	96	2847/516	1229/180	545/67	538/62
Sedl	123	11855/824	3465/266	1465/91	1429/84

Fig. C.1: Experimental results

In the column headed *t_symbols*, two numbers are specified for each grammar. The first number indicates the sum of the sizes of the sets in the range of *t_symbols* and the second number indicates the number of elements in the map. Given a *t_symbols* map, before OP is applied to it, the sets in its range that are identical are merged. In the column headed "merged", the first number specified for each element indicates the sum of the sizes of the sets in the merged map and the second number indicates the number of merged sets. The column headed "Heuristic" shows the length of the resulting string and the number of bases in the partition obtained using the heuristic algorithm described in section 5.2.3.3. The column headed "OP" shows the length of the resulting string and the number of bases obtained using OP.

D. STATISTICS

The table of Figure D.1 shows some statistics about the 6 grammars described earlier and their LALR(k) parser constructed with this method. The information in the table is self-explanatory. In specifying storage values, an 8-bit byte machine is assumed. That is, integer values less than 255 are assumed to be stored in a single byte; integer values that are greater than 255 are assumed to be stored in two bytes.

	Pascal	Pascal1	Pascal2	C	Ada	Sedl
Terminals	63	63	63	86	96	124
Nonterminals	110	111	111	98	304	362
Productions	213	215	215	266	523	746
Storage for Rules	**424**	**428**	**428**	**530**	**1566**	**2235**
Items	626	625	627	786	1527	2398
Scopes	13	13	13	30	25	93
States	193	189	191	206	516	824
Look-ahead states	0	1	5	0	0	0
Shift actions	396	393	393	349	593	1987
Goto actions	336	332	334	975	884	2250
Shift/Reduce actions	329	324	326	987	700	5755
Goto/Reduce actions	574	569	569	322	1779	4360
Reduce actions	326	320	390	1262	1554	4113
Goto entries removed by default	245	244	245	915	705	1763
Goto/Reduce entries removed by default	484	481	481	270	1592	3958
Non-terminals eliminated by default	60	62	61	52	216	206
Length of GOTO_CHECK Table	485	477	480	417	1187	2076
Length of GOTO_INFO Table	374	365	368	318	882	1713
Useful entries in GOTO_INFO	374	365	368	318	882	2076
Storage for compressed GOTO	1455	1431	1440	1251	3561	1713
Terminal states after merging	130	128	133	159	356	513
Shift actions removed by merging	456	450	451	541	679	5484
Reduce actions removed by merging	30	47	48	109	579	1450
Reductions removed by default	263	261	326	1143	908	2544
Length of ACTION_CHECK Table	475	471	490	1076	1133	3014
Length of ACTION_INFO Table	428	420	436	1049	1037	2951
Useful entries in ACTION_INFO	412	420	417	964	1037	2890
Storage for compressed ACTION	1331	407	1362	3174	3207	8916
Storage for Parsing Tables	**2786**	**2742**	**2802**	**4425**	**6768**	**15144**
Actions in Compressed Tables:						
Shifts + Shift/Reduces + LA Shifts	269	268	273	795	614	2258
Gotos + Goto/Reduces	181	176	177	112	366	889
Reduces	55	53	62	90	212	343
Storage for Error maps:						
t_symbols map	399	392	394	928	1651	3188
nt_symbols map	328	323	327	346	1636	2578
SCOPE map	282	280	281	442	780	2843
Required storage for Error maps	**1009**	**995**	**1002**	**1716**	**4067**	**8609**
Unoptimized NAME map	2721	2728	2728	2786	8899	9491
Optimized NAME map	**1832**	**1821**	**1821**	**2477**	**3859**	**6365**
Total Storage Requirement for Parsing and Error Recovery Tables						
(does not include NAME map)	**4704**	**4642**	**4712**	**7088**	**13588**	**28064**

Fig. D.1: Parser Statistics

E. ERROR RECOVERY EXAMPLES

Erroneous Pascal and Ada programs were processed with the error recovery method described in this thesis. The first Pascal program was designed to demonstrate the effectiveness of the scope recovery. The second program was constructed from some interesting examples drawn from [20]. The Ada example is taken from [36]. The Pascal2 grammar of Appendix F and the Ada grammar of [29] were used exactly as is - no user-supplied scopes were used. Thus, the results presented here were obtained automatically from the specification of the input grammars with no intervention, whatsoever, from the user.

Pascal2 Example 1

```
    1. program test(input,output);
    2.     var i,j,k : integer;
    3.         l,m,n : integer;
    4.         x,y,z : real;
    5.         final : real;
    6.         a,b,c : array [1..10] of real;
    7. procedure sub(i,j,k:integer; x,y,z:real);
    8.     var v,w : integer;
    9.     begin
   10.         a := (( 3 ]];
                        ^
                        ^
                          <>
***Error: ")" inserted to complete phrase
***Error: ")" inserted to complete phrase
***Error: Unexpected input discarded
   11.         v := i + j + k;
   12.         w := x + y + z;
                             ^
***Error: "END" inserted to complete phrase started at line 9, column 5

   13.
   14. function f(x,y:real) : result real;
                                      ^
***Error: Unexpected symbol ignored
   15.     var z : real;
   16.     begin
   17.         z := x + y
                         ^
                         ^
***Error: "END" inserted to complete phrase started at line 16, column 5
```

```
***Error: ";" inserted to complete phrase started at line 7, column 1
    18.
    19. begin
    20.     i := 1;
    21.     while i <= 10 do
    22.         begin
    23.         j := 1;
    24.         while j <= 10 do
    25.             begin
    26.             a[i] := a[i] + b[j] + c[j];
    27.             j := j + 1
    28.         end;
    29.         i := i + 1
                         ^
***Error: ; expected after this token
    30.     final := f(x,y,z)*2.0/i+j+345.9;
    31.     while (x = y) do begin
    32.         if y = z then begin
    33.             sub;
    34.             x := ((y + z
                               ^
                               ^
                               ^
                               ^
                               ^
                               ^
                               ^
                               ^
***Error: ")" inserted to complete phrase
***Error: ")" inserted to complete phrase
***Error: "END" inserted to complete phrase started at line 32, column 23
***Error: "ELSE" inserted to complete phrase started at line 32, column 9
***Error: "END" inserted to complete phrase started at line 31, column 22
***Error: "END" inserted to complete phrase started at line 22, column 9
***Error: "END" inserted to complete phrase started at line 19, column 1
***Error: . expected after this token
```

Pascal2 Example 2

```
     1. PROGRAM P(INPUT,OUTPUT);
     2. BEGIN
     3.     FOR K1 := TO NOELEMS DO X:=1
                   ^
                                      ^
***Error: initial_value expected after this token
***Error: ; expected after this token
     4.     IF X=1 THEN
     5.        BEGIN
     6.            WRITELN('0*END OF SORT*')
                                        ^
***Error: "END" inserted to complete phrase started at line 5, column 8

     7.     ELSE
     8.        WRITELN('0***LOOP DETECTED IN INPUT ORDER RELATIONS***');
     9. END.
    10.
    11. PROGRAM HUNTER\INPUT,OUTPUT'?
                      ^
        <--------------------------->
***Error: ( expected instead of this token
***Error: program_heading expected instead
    12.     VAR Q:INTEGER;
    13. BEGIN
    14. END.
    15.
    16. PROGRAM P(INPUT,OUTPUT);
    17.     VAR  I,PRIME,CHECK,NUMB:REAL,A:ARRAY\1..6' OF REAL?
                                            ^
                                                    ^
                                                         ^
                                                                  ^
***Error: ; expected instead of this token
***Error: [ expected instead of this token
***Error: ] expected instead of this token
***Error: ; expected instead of this token
    18. BEGIN
    19. END.
    20.
    21. PROGRAM P(INPUT,OUTPUT);
    22. BEGIN
    23.     FOR I  1 TO 6 DO X:=1
                ^
***Error: := expected after this token
    24. END.
    25.
    26. PROGRAM P(INPUT,OUTPUT);
    27. BEGIN
    28.     CHECK: 1?
```

```
            <------->
***Error: Unexpected input discarded
    29.     BEGIN
    30.        WHILE CHECK)' PRIME DO X:=1
                          <------>
***Error: Unexpected input discarded
    31.     END
    32. END.
    33.
    34. PROGRAM P(INPUT,OUTPUT);
    35.     VAR I:REAL;
    36.     CONST  A[1] 10;A[2] 15;A[3] 25;A[4] 3;?A[5] 50;A[6] 75;
            ^
                      ^
                              ^
                                      ^
                                              ^
                                                  ^
                                                      ^
                                                              ^
***Error: BEGIN expected instead of this token
***Error: := expected after this token
***Error: := expected after this token
***Error: := expected after this token
***Error: := expected after this token
***Error: Unexpected symbol ignored
***Error: := expected after this token
***Error: := expected after this token
    37. BEGIN
    38. END.
          ^
***Error: "END" inserted to complete phrase started at line 36, column 5

    39.
    40. PROGRAM P(INPUT,OUTPUT);
    41.     BEGIN
    42.        IF X<=0 THEN FACT := 1 ENSE FACT := X*FACT(X-1)
                                      ^
***Error: ELSE expected instead of this token
    43.     END;
            ^
***Error: Unexpected symbol ignored
    44. BEGIN
    45. END.
          ^
***Error: "END" inserted to complete phrase started at line 41, column 5

    46.
    47. PROGRAM REID(INPUT,OUTPUT);
    48.     CONST A[1]=10;A[2]=15;A[3]=25;A[4]==35;A[5]=50;A[6]=75;
            ^
```

```
                      ^
                              ^
                                      ^
                                                       ^
                                                               ^
            <----------------------------------------------------->
***Error: BEGIN expected instead of this token
***Error: := expected instead of this token
***Error: := expected instead of this token
***Error: := expected instead of this token
***Error: := expected instead of this token
***Error: := expected instead of this token
***Error: Misplaced construct(s)
    49.     VAR Q:INTEGER;
    50. BEGIN
    51. END.
    52.
    53. PROGRAM P(INPUT,OUTPUT);
    54.     PROCEDURE PR;
    55.     BEGIN
    56.        X:=1;
    57.        BEGIN
    58.            X:=1
    59.        END ;
        ----------->
***Error: Misplaced construct(s) (from line 54, column 5 to ...)

    60.     VAR  FACT, FACT2: REAL ;
    61. BEGIN
    62. END.
    63.
    64. PROGRAM P(INPUT,OUTPUT);
    65. BEGIN;
        <--->
***Error: procedure_or_function_declaration_list expected instead
    66.     PROCEDURE FACTR(N: INTEGER  ; VAR FACTOR : INTEGER ) ;
    67.     BEGIN
    68.         X:=1
    69.     END ;
                ^
***Error: BEGIN expected after this token
    70.     X:=1
    71. END.
```

Ada

```
     1. procedure ATESTS IS
     2.     x: float := 2.1 +;
                            ^
***Error: term expected after this token
     3.     a: array INTEGER range 1..10 of integer;
                   ^
                                       ^
***Error: ( expected after this token
***Error: ) expected after this token
     4.     B: ARRAY [INTEGER RANGE 0..9] OF FLOAT;
                     ^
                                       ^
***Error: ( expected instead of this token
***Error: ) expected instead of this token
     5.     C : ARRAY (BOOLEAN);
                <------------->
***Error: [CONSTANT]constrained_array_definition expected instead
     6.     type t is
     7.         RECORD
     8.             A B: CHARACTER;;
                      ^
                                   ^
***Error: Unexpected symbol ignored
***Error: Unexpected symbol ignored
     9.         END RECORD;
    10.     type b is INTEGER range 1..30;
                      ^
***Error: Unexpected symbol ignored
    11.     subtype c is range 1..30;
                       ^
***Error: type_mark expected after this token
    12.
    13.     procedure count is
    14.         use TEXT_IO;
    15.         x: integer;
                          ^
***Error: BEGIN expected after this token
    16.         GET(x);
    17.         PUT(x);         - - a bad comment
                                <--------------->
***Error: Unexpected input discarded
    18.     end count;
    19.
    20.     procedure q is seperate;
                           ^
***Error: SEPARATE expected instead of this token
    21.
    22.     procedure spell is
    23.       B: ARRAY OF FLOAT;
```

```
                     ^

***Error: index_constraint expected after this token
    24.       x: integer;
    25.       FOR I IN 1 .. 10 LOOP
              ^

***Error: BEGIN inserted before this token
    26.           b(i) := 0.0;
                             ^

***Error: "END LOOP ;" inserted to complete phrase started at line 25, column 7

    27.     end;
    28.
    29.     function DAYS_IN_MONTH(M: MONTH IS_LEAP: BOOLEAN) RETURN DAY IS
                                          ^

***Error: ; expected after this token
    30.     begin
    31.         case M of
                       ^

***Error: ( expected instead of this token
    32.                  FEB => RETURN 28;
        ---------------------->
                                         ^

***Error: Misplaced construct(s) (from line 31, column 9 to ...)

***Error: EXCEPTION expected after this token
    33.             when APR => 30;
                              ^

***Error: RETURN expected after this token
    34.             when SEP | APR | JUN | | NOV => RETURN 30;
                                           ^

***Error: Unexpected symbol ignored
    35.             when others => return 31;
    36.         end case;
                <------->
***Error: Unexpected input discarded
    37.         Z(Y - 5*J + K REM 7) > 0 LOOP
                <---------------------->
***Error: Unexpected input discarded
    38.             x := x + 1;
    39.             go to label;
                    <--->
***Error: Symbols merged to form GOTO
    40.         END LOOP;
    41.         x := x + + 1;
                         ^

***Error: Unexpected symbol ignored
    42.         y := (( 3 ;
                        ^
                        ^

***Error: ")" inserted to complete phrase
***Error: ")" inserted to complete phrase
```

```
    43.         return 28;
    44.         <<label
                      ^
***Error: >> expected after this token
    45.         return 29;
    46.     end of DAYS_IN_MONTH;
                ^
***Error: Unexpected symbol ignored
    47.
    48.     PROCEDURE P IS
    49.         x: integer := 2
                              ^
***Error: ; expected after this token
    50.     begin
    51.         loop
    52.             if x > 0 then y := 2;
    53.             if y < 0 then z := 3;
                                        ^
                                        ^
***Error: "END IF ;" inserted to complete phrase
***Error: "END IF ;" inserted to complete phrase started at line 52, column 13

    54.         end loop;
    55.     end p;
    56.
    57.     procedure test is
    58.         x: array(123.144) of real;
                         <-----^
***Error: ".. simple_expression" inserted to complete phrase
    59.         Y: INTEGER = 5;
                           ^
***Error: := expected instead of this token
    60.     begin
    61.         if x(1) := y then
                        ^
***Error: Invalid relational_operator
    62.             NULL;
    63.         ELSEIF X(2) > Y THEN
                ^
***Error: ELSIF expected instead of this token
    64.             NULL;
    65.         end if;
    66.     end atests;
                      ^
***Error: "BEGIN sequence_of_statements END ;" inserted to complete
          phrase started at line 1, column 1
```

F. PASCAL2 GRAMMAR

This grammar is an LALR(2) grammar for Pascal derived from the grammar in Appendix D of [11]. It is specified in standard BNF format with adjacent symbols separated by at least one blank. The "%empty" symbol denotes the empty string. All reserved words are written in upper case letters. The lexical categories integer_literal, string_literal, etc, are viewed as terminals. The "--" symbol is used to indicate that the rest of the line is a comment. Comments are included wherever there is a deviation from the syntax given in Appendix D.

The information associated with this grammar is broken down into four sections. The first section is a "Terminals" section in which the terminals of the grammar are listed. The second section is a "Rules" section where the rules of the grammar are specified. The third section is a "Names" section where symbols of the grammar are mapped into a different name to be used in issuing error diagnosis. The fourth and final section is a "Scopes" section where the scopes that were computed automatically from the grammar are given. Each scope is written in the form of a BNF rule with a "." in its right-hand side to indicate where the prefix of the scope ends and the suffix begins. Note that a scope does not necessarily correspond to a rule of the grammar. In particular, if the suffix of a scoped rule contains nullable nonterminals in its suffix, these nullable symbols are removed from the scope suffix.

Terminals

```
PROGRAM BEGIN END FUNCTION PROCEDURE VAR TYPE CONST LABEL GOTO WITH
DO FOR TO DOWNTO REPEAT UNTIL WHILE CASE OF IF THEN ELSE RECORD SET
FILE_tok ARRAY PACKED OR IN DIV MOD AND NOT NIL DIRECTIVE
IDENTIFIER      STRING_LITERAL      INTEGER_LITERAL      REAL_LITERAL
+  -  *  /  =  <>  <  <=  >  >=  .  ..  ,  :  ;  :=  (  )  [  ]  ^
```

Rules

```
program_list ::= %empty  |  program_list program
program ::= program_heading block .
program_heading ::= PROGRAM IDENTIFIER ( file_identifier_list ) ;
file_identifier ::= IDENTIFIER
block ::= label_declaration_part constant_definition_part
          type_definition_part variable_declaration_part
          procedure_and_function_declaration_part
          statement_part
label_declaration_part ::= %empty  |  LABEL label_list ;
label ::= integer_literal
constant_definition_part  ::= %empty  |  CONST constant_definition_list ;
```

```
constant_definition ::= IDENTIFIER = constant
constant ::= unsigned_number      |  sign unsigned_number
           | constant_identifier  |  sign constant_identifier
           | string_literal
unsigned_number ::= integer_literal  |  real_literal
sign ::= +  |  -
constant_identifier ::= IDENTIFIER
type_definition_part ::= %empty  |  TYPE type_definition_list ;
type_definition ::= IDENTIFIER = type
type  ::= simple_type  |  structured_type  |  pointer_type
simple_type ::= scalar_type  |  subrange_type  |  type_identifier
scalar_type ::= ( identifier_list )
subrange_type ::= constant .. constant
type_identifier ::= IDENTIFIER
structured_type ::= unpacked_structured_type
                  |  PACKED unpacked_structured_type
unpacked_structured_type ::= array_type  |  record_type
                           | set_type    |  file_type
array_type ::= ARRAY [ index_type_list ] OF component_type
index_type ::= simple_type
component_type ::= type
record_type ::= RECORD field_list END
field_list ::= fixed_part | variant_part  |  fixed_part ; variant_part
fixed_part ::= record_section  |  fixed_part ; record_section
record_section ::= %empty  |  field_identifier_list : type
variant_part ::= CASE tag_field type_identifier OF
                      variant_list
tag_field ::= %empty  |  field_identifier :
field_identifier ::= IDENTIFIER
variant ::= %empty  |  case_label_list : ( field_list )
case_label ::= constant
set_type ::= SET OF base_type
base_type ::= simple_type
file_type ::= FILE OF type
pointer_type ::= ^ type_identifier
variable_declaration_part ::= %empty  |  VAR variable_declaration_list ;
variable_declaration ::= identifier_list : type
procedure_and_function_declaration_part ::= %empty
             | procedure_or_function_declaration_list ;
procedure_or_function_declaration ::= procedure_declaration
                                    | function_declaration
procedure_declaration ::= procedure_heading block
                        | procedure_heading DIRECTIVE
procedure_heading ::= PROCEDURE IDENTIFIER ;
                    | PROCEDURE IDENTIFIER
                          ( formal_parameter_section_list ) ;
formal_parameter_section ::= parameter_group  |  VAR parameter_group
                           | FUNCTION parameter_group
                           | PROCEDURE identifier_list
parameter_group ::= identifier_list : type_identifier
function_declaration ::= function_heading block
```

```
                        |  function_heading DIRECTIVE
    function_heading ::= FUNCTION IDENTIFIER : result_type ;
            | FUNCTION IDENTIFIER
                       ( formal_parameter_section_list ) : result_type ;
    result_type ::= type_identifier
    statement_part ::= compound_statement
    statement ::= unlabelled_statement  |  label : unlabelled_statement
    unlabelled_statement ::= simple_statement  |  structured_statement
    simple_statement ::= assignment_statement  |  procedure_statement
                       | goto_statement        | empty_statement
    assignment_statement ::= variable := expression
--
--  The variable on the left-hand side of the assignment may be a
--  file_identifier, a referenced_variable, a pointer variable,
--  a simple identifier, an indexed variable, a field_designator,
--  a file_buffer or a function identifier.
--
    record_variable ::= variable
    variable ::= IDENTIFIER  | variable [ expression_list ]
               | record_variable . IDENTIFIER
               | variable ^                 -- file or pointer variable
    expression ::= simple_expression
                 | simple_expression relational_operator simple_expression
    relational_operator ::= =  |  <>  |  <  |  <=  |  >=  |  >  | IN
    simple_expression ::= term  |  sign term
                        | simple_expression adding_operator term
    adding_operator ::= +  |  -  |  OR
    term ::= factor  |  term multiplying_operator factor
    multiplying_operator ::= *  |  /  |  DIV  |  MOD  |  AND
    factor ::= variable       | unsigned_constant
             | ( expression ) | function_designator
             | set            | NOT factor
    unsigned_constant ::= unsigned_number  |  string_literal  |  NIL
    function_designator ::= function_identifier ( actual_parameter_list )
    function_identifier ::= IDENTIFIER
    set ::= [ ]  |  [ element_list ]
    element ::= expression  |  expression .. expression
    procedure_statement ::= procedure_identifier
                          | procedure_identifier ( actual_parameter_list )
    procedure_identifier ::= IDENTIFIER
    actual_parameter ::= expression
--
--  An actual parameter may be a simple variable, a procedure
--  identifier or a function identifier.  These classes are
--  included under expression.
--
    goto_statement ::= GOTO label
    empty_statement ::= %empty
    structured_statement ::= compound_statement    |  conditional_statement
                           | repetitive_statement  |  with_statement
    restricted_statement ::= simple_statement
```

```
                           | compound_statement
                           | IF expression THEN
                                  restricted_statement [;]
                             ELSE restricted_statement
                           | case_statement
                           | restricted_while_statement
                           | repeat_statement
                           | restricted_for_statement
                           | restricted_with_statement
    compound_statement ::= BEGIN statement_list END
    conditional_statement ::= if_statement  |  case_statement
    if_statement ::= IF expression THEN
                         statement
                   | IF expression THEN
                         restricted_statement [;]
                     ELSE statement
--
-- Note that the use of [;] in the specification of an if_statement
-- is an extension that allows the parser to accept such a statement
-- with a ";" preceeding the ‘‘ELSE’’.  With this arrangement, a
-- semantic warning message is emitted when the ";" is present.
--
    case_statement ::= CASE expression OF
                            case_list_element_list
                       END
    case_list_element ::= %empty  |  case_label_list : statement
    repetitive_statement ::= while_statement
                           |  repeat_statement
                           |  for_statement
    while_statement ::= WHILE expression DO
                              statement
    restricted_while_statement ::= WHILE expression DO
                                         restricted_statement
    repeat_statement ::= REPEAT
                               statement_list
                         UNTIL expression
    for_statement ::= FOR control_variable := for_list DO
                          statement
    restricted_for_statement ::= FOR control_variable := for_list DO
                                     restricted_statement
    for_list ::= initial_value TO final_value
               | initial_value DOWNTO final_value
    control_variable ::= IDENTIFIER
    initial_value ::= expression
    final_value ::= expression
    with_statement ::= WITH record_variable_list DO
                            statement
    restricted_with_statement ::= WITH record_variable_list DO
                                       restricted_statement
--
-- Expansion of lists
```

```
--
index_type_list ::= index_type  |  index_type_list , index_type
expression_list ::= expression  |  expression_list , expression
identifier_list ::= identifier  |  identifier_list , identifier
field_identifier_list ::= field_identifier
                        | field_identifier_list , field_identifier
file_identifier_list ::= file_identifier
                       | file_identifier_list , file_identifier
label_list ::= label  |  label_list , label
constant_definition_list ::= constant_definition
                           | constant_definition_list ; constant_definition
type_definition_list ::= type_definition
                       | type_definition_list ; type_definition
procedure_or_function_declaration_list ::= procedure_or_function_declaration
                                         | procedure_or_function_declaration_list ;
                                           procedure_or_function_declaration
variant_list ::= variant  |  variant_list ; variant
case_label_list ::= case_label  |  case_label_list , case_label
variable_declaration_list ::= variable_declaration
                            | variable_declaration_list ; variable_declaration
formal_parameter_section_list ::= formal_parameter_section
                | formal_parameter_section_list ; formal_parameter_section
actual_parameter_list ::= actual_parameter
                        | actual_parameter_list , actual_parameter
element_list ::= element  |  element_list , element
statement_list ::= statement  |  statement_list ; statement
case_list_element_list ::= case_list_element
                         | case_list_element_list ; case_list_element
record_variable_list ::= record_variable
                       | record_variable_list , record_variable
--
-- Expand the optional semicolon nonterminal that may appear before ELSE
--
[;] ::= %empty
      | ;         -- Issue a warning when this rule is reduced
```

Names

```
EOFT_SYMBOL -> 'End of Pascal source'
```

Scopes

```
if_statement ::= IF expression THEN restricted_statement [;]  .ELSE
restricted_statement ::= IF expression THEN restricted_statement [;]  .ELSE
block ::= label_declaration_part constant_definition_part
          type_definition_part variable_declaration_part
          procedure_and_function_declaration_part  .statement_part
case_statement ::= CASE expression OF case_list_element_list  .END
variant ::= case_label_list COLON ( field_list  .)
function_designator ::= function_identifier ( actual_parameter_list  .)
variable ::= variable [ expression_list  .]
```

```
repeat_statement ::= REPEAT statement_list  .UNTIL expression
compound_statement ::= BEGIN statement_list  .END
set ::= [ element_list  .]
factor ::= ( expression  .)
record_type ::= RECORD field_list  .END
procedure_and_function_declaration_part ::=
                       procedure_or_function_declaration_list  .;
```

BIBLIOGRAPHY

[1] D. E. Knuth
On the Translation of Languages from Left to Right
Information and Control 8:6, 607-639, 1965

[2] D. R. Fulkerson and D. A. Gross.
Incidence matrices and interval graphs.
Pacific J. Math., 15 (1965), no. 3.

[3] F. L. DeRemer
Practical Translators for LR(k) Languages.
Ph.D. dissertation, MIT, Cambridge, Mass., 1969

[4] F. L. DeRemer
Simple LR(k) Grammars.
Comm. ACM 14, 7, 453-460 July 1971

[5] D. E. Knuth
Top-down syntax analysis.
Acta Informatica 1, 79-110 (1971)

[6] W. R. Lalonde
An Efficient LALR Parser Generator.
Tech. Rep. 2, Computer Systems Research Group, Univ. Toronto, 1971

[7] R. E. Tarjan
Depth-First Search and Linear Graph Algorithms.
SIAM J. Computing 1, 146-160 (1972)

[8] R. M. Karp
Reducibility among combinatorial problems
R. E. Miller and J. W. Thatcher (eds.), Complexity of Computer Computations,
Plenum Press, New York, 85-103. 1972

[9] Alfred V. Aho, Jeffrey D. Ullman
The Theory of Parsing, Translation, and Compiling Volume I & II
Prentice Hall, Inc 1972

[10] S. C. Johnson
YACC - Yet Another Compiler-Compiler.
Tech. Rep. CSTR 32, Bell Labs., Murray Hill, N.J., 1974.

[11] Kathleen Jensen, Niklaus Wirth
Pascal User Manual And Report, Second Edition
Springer-Verlag, 1974

[12] Marc L. Joliat A Simple Technique for Partial Elimination of Unit Productions from LR(k) Parsers.
IEEE Transactions on Computers, 763-764, July 1976

[13] D. Pager
A Practical General Method for Constructing LR(k) Parsers.
Acta Inf. 7, 3 (1977), 249-268.

[14] J. Eve, R. Kurki-Suonio
On Computing the Transitive Closure of a Relation.
Acta Inf. 8 (1977), 303-314.

[15] S. F. Ziegler. Smaller faster table driven parser.
Unpublished manuscript, Madison Academic Computing Center,
University of Wisconsin, Madison, Wisconsin, 1977.

[16] S. Even, D. I. Lichtenstein, Y. Shiloach
Remarks on Ziegler's method for matrix compression.
Unpublished manuscript, 1977

[17] Penello T.J. and DeRemer, F.A.
A Forward Move for LR Error Recovery
Conf. Record ACM Symposium on Principles of Programming Languages,
January, 1978

[18] Mickunas M.D., and Modry, J. A.
Automatic Error Recovery for LR parsers
Comm. ACM 21, 6
(June 1978), 459-465

[19] Ripley, G. D., and Druseikis, F.C.
A statistical analysis of syntax errors
Journal of Computer Languages 3,4 (1978) (227-240)

[20] D. J. Ripley
Pascal Syntax Errors Data Base
RCA Laboratories, Princeton, N.J., Apr 1979

[21] S. L. Graham, C. B. Haley, W. N. Joy
Practical LR Error Recovery
Proceedings of the SIGPLAN 79 Symposium on Compiler Construction
(August 6-10, 1979, Denver) ACM, New York, 1979, pp 168-175.

[22] Michael R. Garey, David S. Johnson.
Computers and Intractability.
W. H. Freeman and Company, 1979

[23] Robert E. Tarjan, Andrew C. Yao.
Storing a sparse Table.
Communications of the ACM 22 606-611, Nov.1979

[24] Bent Bruun Kristensen, Ole Lehrmann Madsen
Methods for Computing LALR(k) Lookahead
ACM Transactions on Programming Languages and Systems,
Vol. 3, No. 1, January 1981, Pages 60-82.

[25] Philippe Charles
Implementation of an LALR Parser Generator
M.S. thesis, NYU, June 1982

[26] Frank DeRemer, Thomas Penello
Efficient Computation of LALR(1) Look-Ahead Sets
ACM Transactions on Programming Languages and Systems,
Vol. 4, No. 4, October 1982, Pages 615-649.

[27] Seppo Sippu, Eljas Soisalon-Soininen
A Syntax-Error-Handling Technique and its Experimental Analysis
ACM Transactions on Programming Languages and Systems, Vol. 5, No. 4,
October 1983, Pages 656-679

[28] Ref. Manual for the ADA Programming Language
ANSI/Mil-STD-1815A-1983, U.S. Dept. of Defense.

[29] Philippe G. Charles, Gerald A. Fisher.
An LALR(1) grammar for ANSI Ada.
Ada Letters vol. 3, No. 4 37-50 (Jan-Feb. 1984).

[30] Peter Dencker, Karl Durre, and Johannes Heuft.
Optimization of Parser Tables for Portable Compilers.
Acm Transactions on Programming Languages and Systems
6,4 (Oct. 1984), 546-572.

[31] Joseph C. H. Park, K. M. Choe, C. H. Chang
A New Analysis of LALR Formalisms
ACM Transactions on Programming Languages and Systems,
Vol. 7, No. 1, January 1985, Pages 159-175.

[32] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers, Principles, Techniques, and Tools
Addison Wesley, 1986

[33] J. T. Schwartz, R. B. B. Dewar, E. Dubinsky, E. Schonberg.
Programming With Sets: An Introduction to SETL
Springer-Verlag, 1986

[34] Thomas J. Sager
A Short Proof of a Conjecture of DeRemer and Pennello
ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986, Pages 264-271

[35] Fred Ives
Unifying View of Recent LALR(1) Lookahead Set Algorithms
SIGPLAN Notices 21, 7 (July 1986), 131-135

[36] Michael Burke, Gerald A. Fisher
A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery
ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2,
April 1987, Pages 164-197

[37] C. Fisher, R. J. LeBlanc
Crafting a Compiler
Benjamin/Cummings, Menlo Park, 1988

[38] Manuel E. Bermudez, George Logothetis
Simple Computation of LALR(1) Lookaheaed Sets
Information Processing Letters 31 (1989) 233-238

[39] Philippe Charles, Laurent Pautet
Efficient Representation of LR Parsing Information for Error Recovery
Unpublished paper, 1989

[40] Philippe Charles
Recent Algorithms for constructing LALR parsers
(Unpublished) Survey paper, 1989

[41] Kirk Snyder
The SETL2 Programming Language
Technical Report 490,
Courant Institute of Mathematical Sciences, New York university, 1990

[42] Jüergen Uhl: Private communications

[43] Franco Gasperoni: Private communications