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.

Approved:_____

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 **3**
- 1.1 Introduction . . . 3
- 1.2 The Problems . . . 4
  - 1.2.1 LALR(k) Parser Construction . . . 5
  - 1.2.2 Error recovery . . . 6
  - 1.2.3 Parsing Tables . . . 6
- 1.3 Contributions of the Thesis . . . 7

2 The Parser Generator **8**
- 2.1 Basic Concepts and Terminology . . . 8
  - 2.1.1 LR(k) parsers . . . 9
  - 2.1.2 LALR(k) parsers . . . 10
- 2.2 Previous Work . . . 13
  - 2.2.1 Kristensen and Madsen . . . 13
  - 2.2.2 DeRemer and Penello . . . 16
  - 2.2.3 Park, Choe and Chang . . . 20
  - 2.2.4 Bermudez and Logothetis . . . 23
- 2.3 Improvements . . . 24
  - 2.3.1 Improving the KM algorithm . . . 24
  - 2.3.2 Improving the DP algorithm . . . 25
  - 2.3.3 Computation of FIRST sets . . . 26
  - 2.3.4 Improving the BL algorithm . . . 27
- 2.4 Remarks . . . 28

3 LALR(k) Lookahead Sets with Varying-Length Strings **29**
- 3.1 LALR(k) Lookahead Sets . . . 35
- 3.2 Computation of READ_{k*}, FOLLOW_k and LALR_k Sets . . . 36
- 3.3 Computation of Varying-Length Lookahead Strings . . . 41
- 3.4 Remarks . . . 44

4 LALR(k) Parser and Error Recovery **48**
- 4.1 Overview . . . 48
  - 4.1.1 The Parsing Framework . . . 48
  - 4.1.2 Error Recovery . . . 48
  - 4.1.3 Error Detection . . . 49

4.2 The Driver . . . 51
4.3 Recovery Strategies . . . 56
4.3.1 Simple Recovery . . . 56
4.3.2 Phrase-Level Recovery . . . 59
4.3.3 Scope Recovery . . . 62
4.4 Recovery Phases . . . 67
4.4.1 Primary Phase . . . 68
4.4.2 Secondary Phase . . . 69
4.4.3 Error Diagnosis . . . 69
4.4.4 Error Repair . . . 71
4.4.5 The Input Buffer . . . 72
4.5 Remarks . . . 74

5 Table Optimization and Generation **75**
5.1 Parsing Tables . . . 76
5.1.1 The Row-Displacement Scheme . . . 77
5.1.2 Transformations . . . 81
5.1.3 Compaction of the Parsing Tables . . . 83
5.2 Error Recovery Tables . . . 89
5.2.1 Optimization of Candidates . . . 89
5.2.2 Optimization of Names . . . 91
5.2.3 Optimization of Finite Subsets of a Small Universe . . . 91
5.3 Remarks . . . 94

A Experimental Results with the LALR Algorithms **97**

B String Compatibility **101**

C Optimal Partition **103**
C.1 Reduction to Maximum Weighted Bipartite Matching Problem . . . 103
C.1.1 Bipartite Graph . . . 103
C.1.2 Weight Function . . . 105
C.2 Complexity . . . 106
C.3 Experimental Results with OP . . . 106

D Statistics **107**

E Error Recovery Examples **109**

F Pascal2 Grammar **117**

Chapter 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 [12] which is also described in detail in [33], 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 [41]. Bermudez and Logothethis, in 1989 [39], 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 [32]; Ives, 1986 [36]. 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] [37], phrase-level (or secondary) recovery [9] [21] [27] [37], and scope recovery [37].

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 [37]. 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.

Chapter 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 [42] 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,

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

.

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

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

2.1.1 LR(k) parsers

An LR(k) item is a quadruple (A, α, β, u), written $[A \rightarrow \alpha \cdot \beta, u]$, where $A \rightarrow \alpha\beta \in P$ 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:

$$\text{CLOSURE}(K) = K \cup \{[B \rightarrow \cdot\gamma, v] \mid v \in \text{FIRST}_k(\beta u), [A \rightarrow \alpha \cdot B\beta, u] \in \text{CLOSURE}(K),\ B \rightarrow \gamma \in P\}$$

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:

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

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:

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

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*: $[S' \rightarrow \cdot S]$; 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 $M_k^G \times V \rightarrow M_k^G$.*

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 $\mathrm{KERNEL}(p)$ is called a kernel item of p. An item in p that is not in $\mathrm{KERNEL}(p)$ 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*

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

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

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

and

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

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

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 $\mathrm{CORE}(\mathrm{IS}_0) = \mathrm{CORE}(\mathrm{IS}_k)$, and that $\mathrm{GOTO}_k(p, X)$, 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, $\mathrm{URCORE}_k(p) \neq \emptyset$, for all $k > 0$ and $p \in M_0$.

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

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

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,*

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

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*

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

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

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

Theorem 2.1.3 *Let $p \in M_k$ then*

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

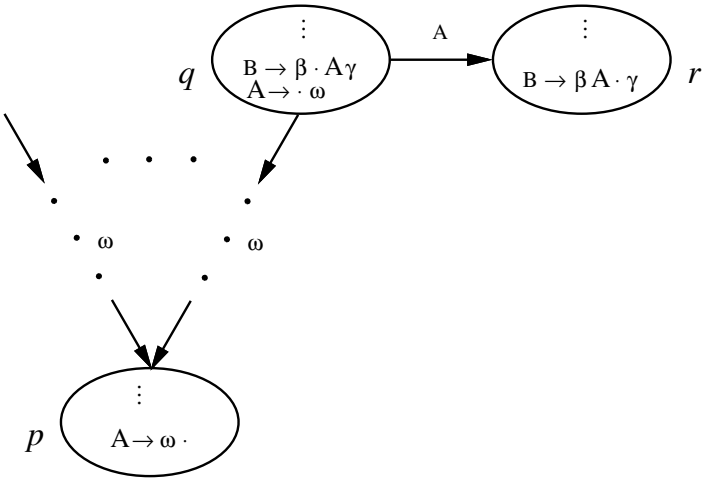

Figure 2.1: Lookahead paths for final item $[A \rightarrow \omega\cdot]$ in state p

Theorem 2.1.4 *Let $p \in M_k$ and let $[A \rightarrow \cdot\alpha] \neq [S' \rightarrow \cdot S\perp^k]$. then*

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

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 $[A \rightarrow \omega\cdot]$ 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 $A \rightarrow \omega$ can be applied in state p. The objective is to compute $\mathrm{LALR}_k(p, [A \rightarrow \omega\cdot])$ 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 $A \rightarrow \omega$ in a state p; i.e., the set $\mathrm{PRED}(p,\omega)$. Each state $q \in \mathrm{PRED}(p,\omega)$ contains the initial item $[A \rightarrow \cdot\omega]$ which was introduced in q through closure by at least one item of the form: $[B \rightarrow \beta \cdot A\gamma]$. After reducing by the rule $A \rightarrow \omega$ 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 $\mathrm{LALR}_k(q, [A \rightarrow \cdot\omega])$ 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:

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

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:

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

In [33], 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] [32] [39] 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*

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

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*

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

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 $\mathrm{FIRST}_1(\delta)$ 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*

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

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

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

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*

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

where

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

```
procedure TRANS(q)
    TM := TM ∪ {q};
    for [B → α· X β] ∈ q loop
        if X ∈ T then
            LA := LA ∪ {X};
        elseif X ⇒* ε and (GOTO0(q, X) ∉ TM) then
            TRANS(GOTO0(q, X));
        end if;
    end loop;
end TRANS;

procedure LALR(p, [A → α · β]);
    DONE := DONE ∪ {(p, [A → α · β])};
    for q ∈ PRED(p, α) loop
        TM := ∅;
        TRANS(GOTO0(q, A));
        for [B → γ·Aδ] ∈ q | δ ⇒* ε and (q, [B → γ·Aδ]) ∉ DONE loop
            LALR(q, [B → γ·Aδ]);
        end loop;
    end loop;
end LALR;

function LALR1(p, [A → α · β]);
    DONE := LA := ∅;
    if A = S' then
        LA := {⊥};
    else LALR(p, [A → α · β]);
    end if;
    return LA;
end LALR1;
```

Figure 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 $B \rightarrow \beta A \gamma$, $\gamma \Rightarrow^* \varepsilon$, 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 $[A \rightarrow \cdot\omega]$ which was introduced by closure. The FOLLOW set of a state p and a nonterminal A is defined as follows:

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

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 $[A \rightarrow \omega\cdot]$ is captured by LA sets which are defined as follows:

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

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

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

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 $\mathrm{LR}(k)$, 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 READ$(p, A) \neq \emptyset$. This conjecture was later proved by Sager [35].

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 two arguments: a node $x \in X$, and an integer variable d which represents the depth of the recursion at which it was invoked for node 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, $N(x)$ is set to the depth of the recursion and $F(x)$ is initialized with $F'(x)$. Next, the algorithm loops through the set of elements y related to x, and for each such y, if the computation

```
proc TRAVERSE(x, d)
    S := S + [x];
    N(x) := d;
    F(x) := F'(x);
    for y ∈ X | x R y loop
        if N(y) = Ω then
            TRAVERSE(y, d+1);
        end if;
        N(x) := N(x) min N(y);
        F(x) := F(x) ∪ F(y);
    end loop;
    if N(x) = d then
        until y = x loop
            pop y from S;
            F(y) := F(x);
            N(y) := ∞;
        end loop;
    end if;
end TRAVERSE;

N := ∅;
S := [ ];
for x ∈ X | N(x) = Ω loop
    TRAVERSE(x, 1);
end loop;
```

Figure 2.3: Digraph algorithm

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 $O(n + m)$. 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

$$\text{PATH}_k(B, C) = \bigcup \{\text{FIRST}_k(\beta_n ... \beta_2 \beta_1) \quad | \quad B_0 = B,\ B_n = C,\ n \geq 0,\ B_i \rightarrow B_{i+1}\beta_{i+1} \in P,\ 0 \leq i < n\}$$

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.

$\text{PATH}_k(B, C)$ is a subset of the lookahead strings or initial prefixes thereof that will appear after a nonterminal C when an item $[A \rightarrow \alpha \cdot B\beta]$ 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 $[A \rightarrow \alpha \cdot X\beta, u]$ be an LR(k) item; then,*

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

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

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

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:

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

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

Given an LR(k) kernel item $[A \rightarrow \alpha \cdot B\beta, u]$, 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

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

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 $p \in M_0$, $[A \rightarrow \alpha_1 \cdot \alpha_2] \in P$, and $A \neq S'$; then*

$$\begin{aligned}&\text{LA}_k([A \rightarrow \alpha_1 \cdot \alpha_2], p) = \\ &\quad \{u \mid u \in \text{PATH}_k(A', A) \oplus_k \text{FIRST}_k(\beta_2) \oplus_k LA_k([B \rightarrow \beta_1 \cdot A'\beta_2], q), \\ &\qquad q \in \text{PRED}(p, \alpha_1),\ A'\ \mathbf{L}^*\ A, [B \rightarrow \beta_1 \cdot A'\beta_2] \in \text{KERNEL}(q)\}\end{aligned}$$

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:

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

where

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

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 [32]) 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 [33] [38], but they can also be computed using the digraph algorithm described earlier [41]. 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*

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

In [33], 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 [39], 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 observing that $\mathrm{LALR}_1(q, [A \rightarrow \cdot\alpha]) = L(q, A)$, (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

```
procedure TRANS(q);
    TM := TM ∪ { q }
    LA := LA ∪ {X | X ∈ T, [B → α·Xβ] ∈ q}
    for [B → α·Xβ] ∈ q loop
        if X ⇒* ε and (GOTO0(q, X) ∉ TM) then
            TRANS(GOTO0(q, X));
        end if;
    end loop;
end TRANS;

procedure LALR(p, [A → α · β]);
    for q ∈ PRED(p, α) | [q, A] ∉ DONE loop
        DONE := DONE ∪ {[q, A]};
        TM := ∅;
        TRANS(GOTO0(q, A));
        for [B → γ·Aδ] ∈ q | δ ⇒* ε loop
            LALR([B → γ·Aδ], q);
        end loop;
    end loop;
end LALR;
```

Figure 2.4: Improved TRANS and LALR procedures

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:

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

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* $\mathrm{GOTO}_0(q, A) = r$ *and* $A \Rightarrow^* \varepsilon$.

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

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

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 $\mathrm{FIRST}(\beta)$ 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 [41]. 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,

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

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 $\text{FIRST}_1(A)$ 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 [33].

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:

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

$\text{DF}(B)$ is simply the subset of $\text{SLR_FOLLOW}(B)$ 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:

$$\text{SLR_FOLLOW}(B) = \{\perp\}, \text{ if } B = S'. \text{ Otherwise,} \tag{2.6}$$

$$\text{SLR_FOLLOW}(B) = \text{DF}(B) \cup \bigcup\{\text{SLR_FOLLOW}(A) \mid B \textbf{ s_includes } A\}. \tag{2.7}$$

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 $[A \rightarrow \alpha B \cdot \beta]$. 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 $\text{FIRST}(\beta)$ for all suffix β following B in the kernel items of q. Similarly, assume that $\text{GOTO}_0^G(p, B) = q$, the computation of $\text{DF}([p : B])$ (equation 2.5) is obtained by forming the union of $\text{FIRST}(\gamma)$ 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) [32], 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) [39], 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.