OBTAINING PRACTICAL VARIANTS OF *LL* (*k*) AND *LR* (*k*) FOR *k* >1

BY SPLITTING THE ATOMIC *k*-TUPLE

A Thesis

Submitted to the Faculty

of

Purdue University

by

Terence John Parr

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 1993

ACKNOWLEDGEMENTS

I wish to acknowledge my parents (James and Peggy) and siblings (James Jr., Jacqueline, and Josephine) throughout my education, without whose support I would never have persevered. I thank Professor Hank Dietz, my advisor, for his encouragement and tolerance over the past five years during my Masters and Doctorate.

I thank Professor Russell Quong for his interest in my research and helping me develop many of the advanced language and parsing ideas in the ANTLR parser generator.

I wish to acknowledge Professors Dave Meyer and Jose Fortes for getting me interested in electrical engineering and encouraging me to attend graduate school. Professor Leah Jamieson deserves thanks for taking a chance on me and supporting my continued studies at Purdue.

I thank Professors Matt O'Keefe and Paul Woodward at the University of Minnesota for providing me with the fellowship to finish my studies. Further, I wish to thank Paul Woodward for giving me the work ''ack.'' I acknowledge Kevin Edgar for whom ''no thanks is too much.''

To the users of the Purdue Compiler-Construction Tool Set and codeveloper, Will Cohen, I extend my gratitude for making all of my research worthwhile.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

ABSTRACT

Parr, Terence John. Ph.D., Purdue University, August 1993. Obtaining Practical Variants of $LL(k)$ and $LR(k)$ for $k>1$ by Splitting the Atomic k-Tuple. Major Professor: Henry G. Dietz.

$LL(k)$ and $LR(k)$ parsers for $k>1$ are well understood in theory, but little work has been done in practice to implement them. This fact arises primarily because traditional lookahead information for $LL(k)$ and $LR(k)$ parsers and their variants is exponentially large in k. Fortunately, this worst case behavior can usually be averted and practical deterministic parsers for $k>1$ may be constructed.

This thesis presents new methods for computing and employing lookahead information. Previously, the lookahead depth, k, was generally fixed and the lookahead information was represented as sets of k-tuples. We demonstrate that by modulating the lookahead depth and by splitting the atomic k-tuple into its constituent components, individual terminals, the lookahead information size can be dramatically reduced in practice. We define a linear approximation to conventional lookahead for $LL(k)$ and $LR(k)$ and their variants that reduces space requirements from an exponential function of k to a linear function. Moreover, this approximation results in deterministic parsing decisions for the majority of cases. By compressing lookahead information to near linear space complexity, we show that deterministic parsing for $k>1$ is practical.

CHAPTER 1 INTRODUCTION

Most computer programs accept phrases from an input language and generate phrases in an output language. These input languages are frequently complicated and their recognizers can represent a considerable programming effort; e.g. programming languages, database interfaces, operating system shells, text processors and even games. Language tools such as parser generators are generally used to construct parsers for these languages. There are a variety of parsing strategies commonly in use, each with different recognition abilities, but all strategies can benefit from large amounts of lookahead — the window into the input stream of symbols, called terminals, that a parser examines to make parsing decisions. This thesis is concerned with extending the recognition strength of automatically generated recognizers by empowering them with more than a single terminal of lookahead.

Parsers typically employ a lookahead depth, *k*, of a single terminal because using deeper lookahead was previously considered intractable — the complexity of grammar analysis and the space complexity of the resulting parsers are exponential in the worst case. Fortunately, the worst case is extremely rare and many deterministic parsing strategies may practically employ lookahead depths of $k>1$.

Deterministic parsers that use more than a single terminal of lookahead, $LL(k)$ and $LR(k)$, have two sources of exponential behavior: the number of parser states is exponential in the size of the grammar and the lookahead information is exponential in *k*. The number of parser states can be reduced to a polynomial function of the grammar size by employing any of the weaker $LL(k)$ or $LR(k)$ variants, e.g. $SLL(k)$ [RoS70], $LALL(k)$ [SiS82,SiS90], $SLR(k)$ [DeR69, DeR71], or $LALR(k)$ [DeR69], while still maintaining reasonable recognition strength. Conventional lookahead information, on the other hand, is inherently exponential in nature as it must be able to represent all possible vocabulary-symbol permutations of length *k*.

To facilitate practical parsers for $k>1$, we must change the way lookahead symbols are employed. Previously, at each change of parser state, parsers examined the next *k* terminals of input regardless of whether all *k* terminals were needed and whether lookahead was needed at all. As a result, each input symbol was inspected exactly *k* times. The fact that decisions rarely need all *k* symbols leads us to the concept that a new type of parser, called an optimal parser, could be constructed that inspected each input symbol at most once. Further, if each symbol is to be examined at most once, the conventional lookahead atomic unit, the *k*-tuple, must be dissolved into its constituent components — the individual terminals themselves. By varying the

lookahead depth and by allowing non-*k*-tuple lookahead comparisons, we have removed the two implicit assumptions that led most researchers to consider parsing, for $k > 1$, impractical.

Our approach to parser construction is based upon new data structures, algorithms, and parser lookahead-decision implementations. We begin by constructing a representation of the grammar, called a grammar lookahead automaton (GLA), that represents the collection of all possible lookahead languages for all grammar positions. The lookahead sequences of depth *k* for a position in the grammar correspond to some subset of the sequences of non-ε edges along the walks of length *k* starting from the associated GLA state. The edges found along the walks can be recorded as deterministic finite automata (DFA's), which we store as child-sibling trees. Consequently, all lookahead computations for any $LL(k)$ or $LR(k)$ variant can be elegantly described as constrained automaton traversals.

Straightforward algorithms for lookahead computation have time and space complexities that are, unfortunately, exponential functions of *k*. We circumvent this intractability in three general ways. First, the lookahead depth, *k*, is modulated according to the actual requirements of the parser decision. Second, an approximation to full lookahead, with potentially linear time and linear space complexity, is used in place of the normal lookahead computation before attempting conventional lookahead computations; these results even can be used to reduce the time to compute full conventional lookahead sets. Third, the results of lookahead computations are cached in order to avoid redundant computations. These approximations, denoted $LL^1(k)$ and $LR^1(k)$, represent a significant departure from the normal view of lookahead computation and parser decision construction. Previously, lookahead was stored as sets of atomic *k*-tuples whereas we consider a terminal to be atomic; e.g., the approximate lookahead computation is a form of compression that yields *k* sets of terminals rather than $O(|T|^k)$ *k*-tuples where $|T|$ is the size of the terminal vocabulary. Approximate lookahead has two direct benefits: The computation of the *k* sets is of polynomial complexity (and potentially linear) in *k* and the lookahead decisions in the resulting parsers have sizes linear in *k*.

Although the various $LL(k)$- and $LR(k)$-based parsers need lookahead of different depths for different grammars and grammar positions, parsing decisions themselves are simple mappings from a domain of terminals or terminal sequences to a range of parser actions; hence, parser lookahead decisions are identical in nature regardless of the parsing strategy. We abstract the notion of a lookahead decision to a relation called **induces** that describes this mapping; as a result, any transformation or implementation of an **induces** relation is equally valid for any parsing strategy. The **induces** relation also isolates the computation of lookahead from the induction of parser actions and the type of action. Testing for parser determinism is accomplished by ensuring that the **induces** relations in all parser states are deterministic.

While $LL(k)$ and $LR(k)$ parser construction is well understood from a theoretical standpoint, little practical work has been done because the implementation of lookahead decisions was previously considered intractable. We concentrate, therefore, on the implementation of parser lookahead states. While the worst-case parser decision size is proportional to the worst-case size

of the lookahead information, $O(|T|^k)$, in general, much can be done to reduce this to a practical size. As with lookahead computations themselves, we apply a hierarchical scheme: First, the lookahead depth, *k*, is modulated to use minimum lookahead; most lookahead states can be resolved with only a single terminal of lookahead ($k=1$), which yields a parser whose lookahead information is mostly linear in size. Second, the linear approximation is attempted before full *k*-tuple lookahead; in the event that $k>1$ lookahead is required, it is generally sufficient to look at the terminals visible at particular lookahead depths rather than *k*-sequences of terminals. Finally, when the linear approximation is insufficient, a hybrid decision composed of the linear approximation plus a few *k*-tuple comparisons is used. By constructing parsers that can dynamically switch between different lookahead depths and comparison structures, parsers with large lookahead buffers become practical. We describe these parsers, which have different state types, as heterogeneous parsers.

Because $LL(k)$ and $LR(k)$ languages cannot be recognized with parsers of polynomial size [HuS78], we choose to demonstrate our approach using a variant that has size proportional to the grammar size. We shall emphasize the $LL(k)$ variant $SLL(k)$ in this thesis because $LL(k)$ grammars may be transformed into structurally equivalent $SLL(k)$ grammars [RoS70] and $SLL(k)$ clearly illustrates the important issues in our approach to parser construction — grammar representation, lookahead information representation, lookahead computation, lookahead decision determinism, and lookahead decision implementation. $LALL(k)$, $LL(k)$, $SLR(k)$, $LALR(k)$ and $LR(k)$ parsers are, however, discussed in Chapter 7.

This thesis is organized as follows: Chapter 2 provides motivation for the use of more than a single terminal of lookahead in deterministic $LL(k)$ and $LR(k)$ parsing, outlines previous work in the area of $LL(k)$ and $LR(k)$ parsing, and introduces optimal parsing, which motivated our dissolution of the atomic *k*-tuple. Chapter 3 details our approach to parser representation and Chapter 4 details our new perspective on lookahead information, lookahead computations, and grammar analysis. Using the methodology in Chapters 3 and 4, Chapter 5 provides a complete description of $SLL^1(k)$ parsers including linear grammar analysis and parser construction. Chapter 6 is similar in form to the format of Chapter 5, but describes $SLL(k)$ completely. The $SLL^1(k)$ linear analysis of Chapter 5 is used to reduce grammar analysis time and to reduce the size of $SLL(k)$ parsers. Chapter 7 completes the thesis by describing the other $LL(k)$ and $LR(k)$ variants. In addition, Chapter 7 generalizes $LL^1(k)$ and $LR^1(k)$ to $LL^m(k)$ and $LR^m(k)$.

CHAPTER 2 MOTIVATION

Almost all theoretical work regarding $LL(k)$ [LeS68] and $LR(k)$ [Knu65] parsers has centered around using more than a single terminal of lookahead ($k>1$) while almost all practical work assumes that a single terminal of lookahead is employed. This is primarily because computing lookahead information needed to make parsing decisions is much more difficult for $k>1$ and the resulting information is exponentially large in the worst case. Nonetheless, we maintain that $k>1$ terminals of lookahead are very useful; future chapters provide a mechanism by which parsers that employ a lookahead depth greater than one can be practically implemented.

This chapter defines the notation used throughout this thesis, provides motivation for the use of large amounts of lookahead, describes what others have done with regards to $LL(k)$ and $LR(k)$ parsing, and presents a utopian view of parsing, called optimal parsing; while trying to reduce the number of lookahead inspections, optimal parsing inspires our view that the k terminals of available lookahead can be examined individually rather than in an atomic k-tuple.

2.1 Terminology

In general, we use the notational conventions of [ASU86] and [SiS88, SiS90] in this thesis for discussing parsing theory.

The input to a parser consists of a sequence of *symbols*, which are merely words from a vocabulary, T. A *string* of symbols is a finite, ordered sequence of symbols; the empty string is denoted ε. The set of all strings that may be constructed from a vocabulary, therefore, is T^*, the closure of T. T^+, the set of nonempty strings constructed from T, is $T^* - \varepsilon$, the positive closure of T. A symbol or set of symbols may be raised to an exponent, n, which implies that n of the symbol or symbol are required; e.g. T^3 indicates a sequence of 3 symbols from set T and a^5 indicates *aaaaa*. The prefix of a string, $w = a_1 ... a_n$, is

$$k{:}w = \begin{cases} w & |w| \le k \\ a_1 ... a_k & |w| > k \end{cases}$$

where $|w|$ is the length (number of symbols) of w.

The structure of an input language is described with a *context-free grammar* (CFG) or grammar for short as only *context-free languages* (CFL's) are discussed in this thesis. A CFG is a four-tuple $G = (N, T, P, S)$ where

N	is the finite set of nonterminal symbols
T	is the finite set of terminal symbols (input language vocabulary)
$P \subseteq N \times (N \bigcup T)^*$	is the finite set of productions of the form $A \rightarrow \alpha$ where $\alpha \in (N \bigcup T)^*$
$S \in N$	is the start symbol

where we augment all T with ''$'', the end-of-file marker. Define an *item* to be a position in the grammar denoted $[A \rightarrow \alpha \bullet \beta]$ for some $\alpha, \beta \in (N \bigcup T)^*$. We further define the left edge of a production to be an item of the form $[A \rightarrow \bullet \alpha]$; the brackets will be left off when it is obvious that an item is being discussed. The grammar vocabulary is $V = N \bigcup T$, whereas the vocabulary of the input language is T. A *rule* is a nonterminal plus a collection of one or more of its productions.

The size of an object is denoted $|...|$. For example, the size of a grammar $|G|$ is

$$|G| = \sum_{A \rightarrow \alpha \in P} |A\alpha|$$

which is the number of distinct positions within the right hand sides of all productions; similarly, $|T|$, $|N|$, and $|P|$ are the number of terminals, nonterminals and productions, respectively.

Unless otherwise specified, the lower-case Greek letters, α, β, ..., ζ represent strings of grammar symbols. The lower-case Latin letters u, v, ..., z represent strings of terminals while most lower-case letters appearing earlier in the alphabet are single terminals; the letters i–q represent action numbers, integers or automaton states. Upper-case Latin letters generally represent nonterminals. Define the derivation relations $\Rightarrow$, $\Rightarrow^*$ and $\Rightarrow^+$ as ''directly derives,'' ''derives in zero or more steps,'' and ''derives in one or more steps'' where each relation may be annotated with *lm* for a leftmost or *rm* for a rightmost derivation; i.e.

$uA\beta \Rightarrow_{lm} u\alpha\beta$	leftmost derivation
$\beta Au \Rightarrow_{rm} \beta\alpha u$	rightmost derivation

with $\alpha, \beta \in V^*$, $u \in T^*$ and $A \rightarrow \alpha \in P$.

The language generated by a grammar, $L(G)$, is the set of all strings derivable from the start symbol; i.e. $L(G) = \{ u \in T^* \mid S \Rightarrow^+ u \}$. In general, when $S \Rightarrow^* \alpha$, α is called a *sentential form* of G; if $\alpha \in T^*$, such as u above, α is called a *sentence* of G. A *left (right) sentential form* is a sentential form resulting from a leftmost (rightmost) derivation. Unless specified otherwise, the start symbol is always that of the first production given.

$FIRST_k(\alpha)$ is the set of all strings, less than or equal to k in length, that can begin any sentence derived from $\alpha \in T^*$:

$$FIRST_k(\alpha) = \{\ k{:}w \mid \alpha \Rightarrow^* w \text{ where } w \in T^*,\ \alpha \in V^*\ \}$$

If α derives the empty string (ε) then $FIRST_k$ is not ε as many propose and if α is itself ε then $FIRST_k$ is the empty set, $\varnothing$. Occasionally, ε is used as an imaginary placeholder terminal for implementation's sake, but should not be a result of a $FIRST_k$ operation.

The computation $FOLLOW_k(A)$, for some $A \in N$, is the set of all strings which can be matched immediately following the application of A in any valid derivation; i.e.

$$FOLLOW_k(A) = \{\ FIRST_k(\beta) \mid S \Rightarrow^* \alpha A \beta\ \}$$

for some $\alpha \in T^*$, $\beta \in V^*$ for $LL(k)$ and $\alpha \in V^*$, $\beta \in T^*$ for $LR(k)$.

$LL(k)$ [LeS68] and $LR(k)$ [Knu65] parsers recognize $LL(k)$ and $LR(k)$ languages described by $LL(k)$ and $LR(k)$ grammars, respectively. An $LL(k)$ grammar is one for which each nonterminal, A, satisfies the $LL(k)$ condition:

$$FIRST_k(\alpha_1 \delta) \bigcap FIRST_k(\alpha_2 \delta) = \varnothing$$

for all left sentential forms $uA\delta$ and distinct productions $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$ in P. A Strong $LL(k)$ ($SLL(k)$) grammar [RoS70] is defined similarly:

$$FIRST_k(\alpha_1 FOLLOW_k(A)) \bigcap FIRST_k(\alpha_2 FOLLOW_k(A)) = \varnothing$$

A grammar is $LR(k)$ if its canonical $LR(k)$ parser is deterministic. Equivalently, [SiS90] states that the conditions

$$S \Rightarrow \delta_1 A_1 y_1 \Rightarrow \delta_1 \omega_1 y_1 = vy_1$$
$$S \Rightarrow \delta_2 A_2 y_2 \Rightarrow \delta_2 \omega_2 y_2 = vxy_2$$

always imply that $\delta_1 = \delta_2$, $A_1 = A_2$, and $\omega_1 = \omega_2$. A Simple $LR(k)$ ($SLR(k)$) [DeR71] grammar is one for which the $SLR(k)$ parser is deterministic. [SiS90] states that an $SLR(k)$ parser is deterministic if for all states q, the following hold:

(1) Whenever q contains a pair of distinct items $[A_1 \rightarrow \omega_1 \bullet]$, $[A_2 \rightarrow \omega_2 \bullet]$, then
$FOLLOW_k(A_1) \bigcap FOLLOW_k(A_2) = \varnothing$

(2) Whenever q contains a pair of items $[A_1 \rightarrow \alpha \bullet a\beta]$, $[B \rightarrow \omega \bullet]$, where a is a terminal, then
$FIRST_k(a\beta FOLLOW_k(A)) \bigcap FOLLOW_k(B) = \varnothing$

We shall refer to any state that examines lookahead within a parser as a *parser decision state*. Also, the k terminal symbols in a deterministic parser's lookahead buffer are referred to as $\tau_1, ..., \tau_k$. The reader is assumed to be generally familiar with the actual construction of ''top-down'' ($LL(k)$) and ''bottom-up'' ($LR(k)$) parsers; see [SiS90] or [ASU86] for detailed discussions.

In [SiS90], the authors use the notation $C(k)$ to refer generically to $LL(k)$, $LR(k)$ and their variants. We will use this notation as well when referring to generic deterministic parsers.

2.2 The Need for k Terminals of Lookahead

Consider modifying a compiler whose recognizer was built using a parser generator. A change of target machine could involve changing semantic actions within the grammar and probably the placement of some of these actions. Because recognition and translation are logically separate phases, one phase should not interfere with the behavior of the other. For a semantic modification to break the recognizer is unacceptable, but this is precisely what can occur with an *LR*-based parser generator because productions must be ''cracked'' to create reductions corresponding to the action positions. Placing an action within an *LR* grammar implies rule cracking, thus, we use both notations interchangeably in this section. Placing actions within an $LALR(1)$ [DeR71] parser generator, such as YACC [Joh78], can be a terribly irritating experience, but has become acceptable due to its apparent unavoidability. This is because the only commonly available substitute was $LL(1)$. Although $LL(1)$ parsers are not sensitive to action placement and are more flexible semantically, in practice, $LL(1)$ recognizers are noticeably weaker $LALR(1)$; $LL(1)$-conforming grammars are more difficult to construct.

Can sensitivity to action placement be reduced while retaining reasonable recognition strength? *LR* parsers will never be insensitive to action placement, but both *LL* and *LR* techniques can benefit from k terminals of lookahead. It is well known that $LL(k)$ is stronger than $LL(k-1)$ [FiL88]. Knuth [Knu65], in contrast, shows that an $LR(k)$ grammar always has an $LR(1)$ equivalent; unfortunately, transforming $LR(k)$ grammars to $LR(1)$ is not easy and, further, grammars cannot be arbitrarily rewritten when actions may be placed anywhere among the

productions. To demonstrate the effect of action placement, this section shows, among other things, that $LL(k)$ is strictly stronger than $LR(k-1)$ given that actions can be placed at any position in a grammar. We suggest that, regardless of the parsing method, $k>1$ terminals of lookahead are useful and that, unfortunately, in the worst-case action placement scenario, $LL(k)$ is the largest class of languages one can recognize with a context-free grammar after the actions have been forced to a production right edge.

This section presents a number of theorems regarding the relative recognition strength of *LL*, *LR*, *LALR* and *SLR* [DeR69, DeR71] grammars augmented with semantic user-defined actions. Section 2.2.1 begins by discussing how actions may be embedded within *LR* productions and proceeds to prove a theorem which is fundamental to our discussion of expressive strength — the strength of $LR(k)$ can be reduced to that of $LL(k)$ by an appropriate choice of action placement. In addition, this section shows that, in fact, $LR(k)$ grammars generate a larger class of languages than $LR(k-1)$ grammars generate when augmented with actions. Section 2.2.2 explores the relationship between the $LR(k)$-based classes of languages and the $LL(k)$ class.

2.2.1 The Effect of Action Placement upon $LR(k)$ and $LL(k)$

In the course of deriving the properties of ‘‘translation grammars,’’ Brosgol [Bro74] made a useful observation about the relationship between $LL(k)$ and $LR(k)$. He showed that a grammar is $LL(k)$ *iff* that grammar, augmented on each left edge with a reference to a unique ε-nonterminal, is $LR(k)$. Because insertion of an action at the left edge of a production implies cracking to introduce such a reference, Brosgol’s work can be seen as proving that, in the worst-case action placement, $LR(k)$ is equivalent to $LL(k)$.

In this section, we examine the more general properties of different parsing methods and lookahead depths relative to placement of actions. We show that $LL(k)$ is insensitive to action placement, $LR(k)$ cannot generally be rewritten as $LR(k-1)$ (when actions may be arbitrarily inserted), and finally that $LL(k)$ is stronger than any deterministic parsing method with less lookahead because of the deleterious effect of arbitrary action-placement upon *LR*’s recognition strength. We begin by defining how an *LR* grammar can be augmented with arbitrarily-placed actions.

Definition: *Rule cracking is the process by which semantic actions may be embedded within LR productions. Productions of the form* $A \rightarrow \alpha @ \beta$ *are translated into* $A \rightarrow A^{(1)} \beta$ *and* $A^{(1)} \rightarrow \alpha @$ *where* $A^{(1)}$ *is unique and* $\alpha, \beta \in V^*$*; @ represents a unique semantic action; T is the set of terminals and N is the set of nonterminals*.

Once a rule has been cracked to force actions to right edges, the actions can be ignored; they do not affect grammar analysis. For example,

$$A \rightarrow A^{(1)}\ \alpha$$
$$A^{(1)} \rightarrow @$$

is identical to

$$A \rightarrow A^{(1)}\ \alpha$$
$$A^{(1)} \rightarrow$$

from a grammar analysis point of view. The productions cannot be recombined to $A \rightarrow \alpha$, however, because at parser run-time the actions must be executed.

We shall present a proof, different from Brosgol's, that *LR*'s strength can be reduced to that of *LL*, but beforehand, consider an *LR* grammar in which actions have been placed at all possible positions in all productions. The augmentation effectively forces the *LR* parser to assume a one-to-one correspondence between parser state and grammar position. *LR*'s recognition strength comes from its ability to be at more than one position in the grammar at once. If this advantage is taken away, the *LR* parser would have a unique mapping from parser state to grammar position; i.e. the *LR* parser could only recognize *LL* languages. Similarly, by placing actions at the left edge of productions, the *LR* parser is forced to know its position in the grammar at the left edge of every rule rather than at the right edge; which implies that it must predict which production it will apply. Once again, this renders the *LR* parser only as strong as *LL*.

Theorem 2.1: *Let G be an* $LR(k)$ *grammar with productions of the form* $A \rightarrow \alpha$. *Construct a new grammar, G', by augmenting productions of G with unique semantic actions,* $@_i$, *such that productions of G' are of the form* $A \rightarrow @_i\ \alpha$. *Crack the productions in G' to force the actions to production right edges and then remove the actions for the sake of grammar analysis.* $G \in LL(k) \Leftrightarrow G' \in LR(k)$.

Proof:

To show that $G' \in LR(k) \Rightarrow G \in LL(k)$, we show that left recursion in G is illegal and that k terminals of lookahead are sufficient to predict each production. Three cases must be considered:

Case (i): Nonterminals in G with only one production, $A \rightarrow \alpha$. After augmentation, nonterminal A in G' will be of the form:

$$A \rightarrow @\ \alpha$$

which is cracked to form:

$$A \rightarrow A^{(1)}\ \alpha$$
$$A^{(1)} \rightarrow @$$

Because $G \in LR(k)$, nonterminal A cannot be left recursive as productions of the form, $A \rightarrow A\ \delta$ (where $\delta \in (N \bigcup T)^*$ with N the set of nonterminals and T the set of terminals) never derive any terminal strings. Because no parsing decision is required to predict the single production of A and because A is not left recursive, the original nonterminal A in G is $LL(k)$-decidable (in fact, A is $LL(0)$).

Case (ii): Nonterminals in *G* with exactly two productions are augmented to form:

$$A \rightarrow @_1\ \alpha$$
$$A \rightarrow @_2\ \beta$$

Productions in *A* are cracked to force the actions, $@_i$, to production right edges. This yields the corresponding portion of *G'*:

$$A \rightarrow A^{(1)}\ \alpha$$
$$A \rightarrow A^{(2)}\ \beta$$
$$A^{(1)} \rightarrow @_1$$
$$A^{(2)} \rightarrow @_2$$

The transformation to *G'* preserves the *LR* condition if there does not exist a *k*-sequence that is common to the lookahead sequences for $A^{(1)}$ and $A^{(2)}$. This indicates that *k* terminals must be sufficient to predict the productions of *A* in *G'* and, hence, to predict productions of *A* in *G*. α and β are not left recursive; if both were left recursive, nonterminal *A* would never derive any terminal string and if one of them were left recursive, a shift/reduce error would occur in the augmented grammar *G'*. To visualize the conjecture that the lookahead sets must be different for the two productions, consider Figure 2.1.

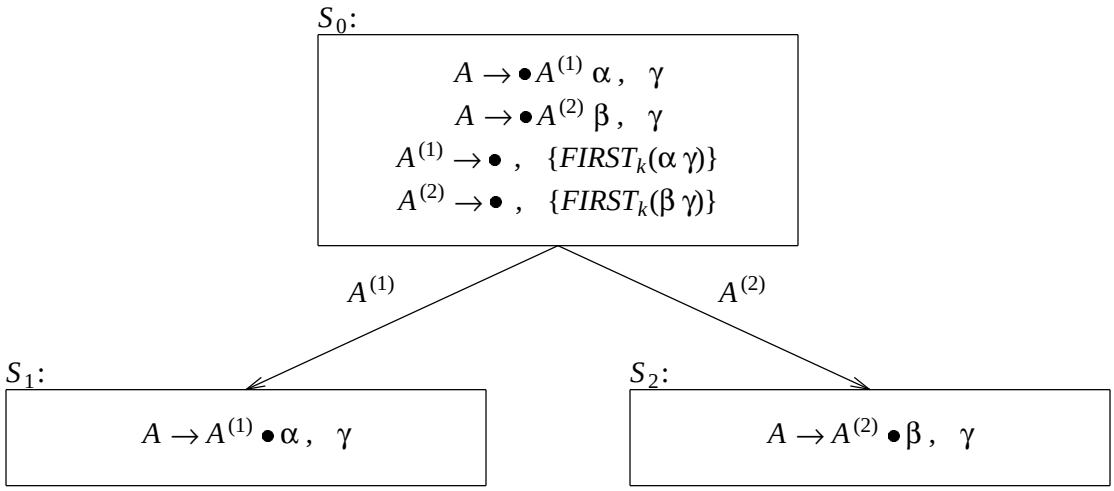


Figure 2.1 Partial $LR(k)$ Machine

The general form of an $LR(k)$ item is $[A \rightarrow \alpha, \gamma]$ [ASU86] with γ the lookahead component ''inherited'' from the state pointing to state S_0 and other items in state S_0. Because *G'* is assumed to be $LR(k)$, the other items do not need to be considered; they, by assumption, do not conflict with the items associated with *A*. If $FIRST_k(\alpha\ \gamma) \cap FIRST_k(\beta\ \gamma) \neq \varnothing$ then the automaton would be unable to decide whether to reduce $A^{(1)}$ or $A^{(2)}$, rendering *G'* non-$LR(k)$. The partial

parsing table for the automaton, Table 2.1, illustrates that if $FIRST_k(\alpha\ \gamma)$ and $FIRST_k(\beta\ \gamma)$ were to overlap, a reduce/reduce conflict would arise.

Table 2.1 Partial Parsing Table

State	Action		Goto	
	$FIRST_k(\alpha\ \gamma)$	$FIRST_k(\beta\ \gamma)$	$A^{(1)}$	$A^{(2)}$
0	r$A^{(1)}$	r$A^{(2)}$	1	2
1	s3			
2		s4		

The symbol r$A^{(1)}$ means *reduce* $A^{(1)}$ and s3 means *shift and go to state 3* [ASU86]. Since the assumption of $G' \in LR(k)$ implies that $FIRST_k(\alpha\ \gamma)$ and $FIRST_k(\beta\ \gamma)$ have no common k-tuples, k terminals of lookahead are sufficient to predict which production of A in G to apply. As mentioned above, α, β in G' must not be left recursive in order to leave $G' \in LR(k)$. Hence, since α and β are not affected by the transformation from G to G', the original nonterminal A must not be left recursive.

When $G' \in LR(k)$, each nonterminal A in G with exactly two productions is $LL(k)$-decidable.

Case (iii): Nonterminals, A, in G with more than two productions:

$$A \rightarrow \alpha$$
$$A \rightarrow \beta$$
$$\ldots$$
$$A \rightarrow \zeta$$

can be transformed into a series of nonterminals with at most two productions by rewriting them in the following way:

$$A \rightarrow \alpha$$
$$A \rightarrow A^{(1)}$$
$$A^{(1)} \rightarrow \beta$$
$$A^{(1)} \rightarrow A^{(2)}$$
$$\ldots$$
$$A^{(m)} \rightarrow \zeta$$

This transformation does not affect the $LR(k)$ nature of G. The results of cases *(i)* and *(ii)* may be used because the $A^{(i)}$ have at most two productions; hence, nonterminal A and all subsequent $A^{(i)}$ created in this fashion are $LL(k)$-decidable.

Cases *(i)*, *(ii)* and *(iii)* indicate that, when $G' \in LR(k)$, each nonterminal in G is $LL(k)$-decidable; therefore, $G \in LL(k)$.

To show that $G \in LL(k) \Rightarrow G' \in LR(k)$, it is sufficient to observe that, by lemma 2.1, $G \in LL(k) \Rightarrow G' \in LL(k)$. Then, $G' \in LR(k)$ since $LL(k) \subset LR(k)$ [RoS70].

□

Theorem 2.1 directly shows that, in the worst case, an $LR(k)$ grammar can be rendered only as expressive as an $LL(k)$ grammar. This result follows from the forced bijection between parser state and grammar position at the point of action insertion.

The same action insertion that weakens *LR* grammars does not affect the nature of an *LL* grammar — even if the productions are cracked in the *LR*-fashion to force actions to a production right edge. We now show that the introduction of unique ε-nonterminals at the left edge of all productions in an $LL(k)$ grammar has no effect upon its $LL(k)$ nature; i.e. the transition from $G \leftrightarrow G'$, as described above, does not affect $LL(k)$ determinism.

Lemma 2.1: *Construct grammar G' from G as before in theorem 2.1.* $G \in LL(k) \Rightarrow G' \in LL(k)$.

Proof:

All nonterminals, A, in G are of the form:

$$A \rightarrow \alpha_1$$
$$A \rightarrow \alpha_2$$
$$\ldots$$
$$A \rightarrow \alpha_n$$

where $\alpha \in T$ are augmented in G' to form:

$$A \rightarrow A^{(1)}\ \alpha_1$$
$$A \rightarrow A^{(2)}\ \alpha_2$$
$$\ldots$$
$$A \rightarrow A^{(n)}\ \alpha_n$$
$$A^{(1)} \rightarrow$$
$$A^{(2)} \rightarrow$$
$$\ldots$$
$$A^{(n)} \rightarrow$$

Because nonterminals $A^{(i)}$ have only one production, they are trivially $LL(k)$. Because, for the augmentation of a production $A \rightarrow \alpha_i$ to $A \rightarrow A^{(i)}\ \alpha_i$, $FIRST_k(A^{(i)}\ \alpha_i\ \gamma) = FIRST_k(\alpha_i\ \gamma)$, the transformation to G' does not affect the lookahead set for productions; γ is derived from sentences of the form $\omega A \gamma$ such that $S \Rightarrow^*_{lm} \omega A \gamma$ with $\omega \in T^*$, $\gamma \in V^*$. Hence, nonterminals that are $LL(k)$ in G are $LL(k)$ in G'. Finally, because all nonterminals A and $A^{(i)}$ in G' are $LL(k)$, $G' \in LL(k)$.

□

Lemma 2.1 shows rule cracking in $LL(k)$ grammars, as performed in Theorem 2.1 for $LR(k)$ grammars, has no effect upon $LL(k)$ determinism. It is also the case that rule cracking for arbitrarily-placed actions has no effect upon the $LL(k)$ nature of a grammar. Informally, rule cracking for actions not at the left edge, $A \rightarrow \alpha\ @\ \beta$, introduces unique nonterminals, $A^{(i)} \rightarrow \alpha$, with only one production; a construct which is clearly $LL(k)$. The cracked rule that invokes $A^{(i)}$ is of the form $A \rightarrow A^{(i)}\ \beta$ and has an unperturbed lookahead set.

In contrast, the placement of actions within a grammar does restrict how a grammar can be *rewritten*. The proof that $LR(1)$ is equivalent to $LR(k)$ in [Knu65] relied upon the fact that grammars could be arbitrarily rewritten — or, more precisely, Knuth's proof did not consider the case where a specific sequence of semantic actions must be ''folded'' into the sequence of terminal matches. When this ability to arbitrarily rewrite a grammar is removed, or when an exact sequence of actions must be triggered relative to the terminal matches, $LR(k)$ is stronger than $LR(1)$. This should not surprise the reader in that, under the stated conditions, we have shown that $LR(k)$ is equivalent to $LL(k)$ and $LL(k)$ is known to be stronger than $LL(k-1)$ [RoS70]. We formalize this notion in the following theorem.

Theorem 2.2: *When actions may be placed arbitrarily among the productions of a grammar, $LR(k)$ grammars cannot, in general, be rewritten to be $LR(k-1)$; i.e. $LR(k-1) \subset LR(k)$ for $k>1$.*

Proof:

We prove that $LR(k)$ is stronger than $LR(k-1)$ by showing that there exists at least one grammar, augmented with actions, which is $LR(k)$ that cannot be rewritten to be $LR(k-1)$ for some k. Consider Grammar G2.1 that is $LR(2)$, before cracking, where x and y are terminals.

$$\begin{aligned} A &\rightarrow @_1\ x\ y \\ A &\rightarrow @_2\ x\ z \end{aligned} \qquad \text{G2.1}$$

Nonterminal A must be cracked to form Grammar G2.2:

$$\begin{aligned} A &\rightarrow A^{(1)}\ x\ y \\ A &\rightarrow A^{(2)}\ x\ z \\ A^{(1)} &\rightarrow @_1 \\ A^{(2)} &\rightarrow @_2 \end{aligned} \qquad \text{G2.2}$$

Clearly, this grammar can be rewritten an infinite number of ways. But, to preserve the semantics of the translation and the syntax of the language, the order of action execution and terminal recognition must be preserved; e.g. $@_1$ cannot be moved to the position between the x and y in production one because the actions $@_i$ must be executed before x has been recognized. Grammar G2.2 generates a language with two sentences, $x\ y$ and $x\ z$, with the constraint that a reduce, either $A^{(1)}$ or $A^{(2)}$, must occur before a sentence is recognized. In order to reduce the correct ε-production, two terminals of lookahead are required to predict which sentence will be recognized. No matter how the grammar is rewritten, as long as it satisfies the constraints mentioned above, entire sentences must be seen to avoid a reduce/reduce conflict relative to $A^{(1)}$ and $A^{(2)}$. This fact

implies that one terminal of lookahead always will be insufficient because two terminals are required to uniquely identify a sentence; Grammars G2.1 and G2.2 can be rewritten to be $LR(1)$. There exists at least one $LR(k)$ grammar that cannot be rewritten to be $LR(k-1)$. Therefore, $LR(k-1) \subset LR(k)$ for $k>1$ when actions can be arbitrarily placed among the productions of a grammar.

□

In effect, Theorem 2.2 shows that $LR(k)$ is not equivalent to $LR(1)$ because embedded actions place severe constraints upon how a grammar may be rewritten. From the point of view of action triggering relative to terminal matching, Theorem 2.2 shows that there exists a language which is $LR(k)$ with no $LR(k-1)$ equivalent that preserves the action-triggering, terminal-matching, sequence. Although there may be many grammars with actions that can be rewritten as $LR(1)$, in general, a $LR(k)$ grammar with actions cannot be rewritten in $LR(1)$ form.

We conclude that $LL(k)$ is stronger than any known deterministic parsing strategy with less lookahead, by combining the results of Theorems 2.2 and Corollary 2.1, given the constraint that actions may be placed anywhere within the associated grammar. Corollary 2.2 states this formally:

Corollary 2.1: $LR(k-1) \subset LL(k)$ *when actions may be placed arbitrarily among the productions of a grammar.*

Proof:
Let G' be a grammar whose productions have been cracked to allow a set of arbitrarily placed actions. $G' \in LL(k) \Rightarrow G' \in LR(k)$ and, by Theorem 2.2, $LR(k-1) \subset LR(k)$, which implies $G' \notin LR(k-1)$ by transitivity. Therefore, there are grammars, G', which are $LL(k)$, but not $LR(k-1)$.

□

The reader may argue at this point that, in practice, there are $LR(k)$ and even $LR(k-1)$ grammars with actions that are not $LL(k)$. While this is correct, we consider the general case of grammars with actions interspersed arbitrarily among the productions. Our assumption allows us to always choose actions on the left edge. We do not suggest that *LL* is as strong as *LR* in practice, we merely show that $k>1$ terminals of lookahead are very useful due to the deleterious effect of action placement on recognition strength. For example, Grammar G2.1 is $LR(0)$ without actions and $LR(2)$ with actions.

This section provided a proof that $LR(k)$ grammars are not always more expressive than $LL(k)$ grammars due to the introduction of extra nonterminals resulting from rule cracking. Lemma 2.1 supported a step in the proof of theorem 2.1, but also suggests the notion that $LL(k)$ grammar analysis is not affected by the introduction of actions. Theorems 2.2 and Corollary 2.1 show that $LR(1)$ is not equivalent to $LR(k)$ when actions can be introduced arbitrarily and that, unfortunately, $LL(k)$ is the largest class of languages that can be generated with a context-free grammar augmented arbitrarily with actions; hence, using $k>1$ terminals of lookahead greatly increases the strength of a particular parsing method. For completeness, in the next section, we

explore how the *LR* derivatives compare to *LL*.

2.2.2 *LALR* and *SLR* Versus *LL*

The results of the previous section provide the framework for re-examining the relationship between *LR*-derivative grammars and *LL* grammars when actions may be placed arbitrarily among the productions. As one might expect, because *LL* and *LR* are equally strong for translation purposes, the *LR* derivatives, which are weaker than *LR*, are weaker than *LL*. This section we extend Brosgol's work by showing that $LALR(k) \subset LL(k)$ and $SLR(k) \subset LL(k)$ whereas, without actions, no strict ordering is observed.

Because $LALR(1)$ is a subset of $LR(1)$, one can observe that if a grammar *G*, augmented with actions on all production left-edges, yielding *G'*, is $LALR(1)$, the original grammar *G* must be $LL(1)$. The following corollary states this supposition formally.

Corollary 2.2: *$G' \in LALR(k)$ is a sufficient condition for the corresponding grammar, G, to be $LL(k)$; where G' is constructed as in theorem 2.1.*

Proof:
Since $LALR(k) \subset LR(k)$ [ASU86], if *G'* is $LALR(k)$ then it is also $LR(k)$. $G' \in LR(k) \Rightarrow G \in LL(k)$.
□

Note that the opposite of corollary 2.2 is not true; $G \in LL(k)$ does not imply $G' \in LALR(k)$. *G* may still be $LL(k)$ even if *G'* is not $LALR(k)$ because there is no strict ordering between $LALR(1)$ and $LL(1)$ for grammars without actions. For example, the following grammar, in YACC notation, is $LL(1)$, but not $LALR(1)$.

$$
\begin{aligned}
&S \rightarrow aA \\
&S \rightarrow bB \\
&A \rightarrow Ca \\
&A \rightarrow Db \\
&B \rightarrow Cb \\
&B \rightarrow Da \\
&C \rightarrow E \\
&D \rightarrow E \\
&E \rightarrow
\end{aligned}
\qquad \text{G2.3}
$$

The problems for $LALR(1)$ arise from the permutations of *C*, *D*, *a*, and *b* in rules *A* and *B* that make $FOLLOW_1(C)$ and $FOLLOW_1(D)$ the same; therefore, lookahead cannot be used to guide the parser when reducing *C* and *D*. Once the lookahead has been effectively removed from consideration, the parser must rely upon context (state) information. Unfortunately, the grammar is not $LALR(0)$; this fact, combined with the $FOLLOW_1$ overlap, renders the grammar non-

LALR (1). A reduce/reduce conflict between rules *C* and *D* is unavoidable.

Corollary 2.2 states that all *LALR* (*k*) grammars, with actions arbitrarily interspersed among the productions, are *LL* (*k*). Further, there are grammars which are *LL* (*k*), but not *LALR* (*k*). Consequently, one may state that *LL* (*k*) is strictly stronger than *LALR* (*k*).

Corollary 2.3: $LALR(k) \subset LL(k)$ *when actions may be placed arbitrarily among grammar productions*.

Proof:
Let *G'* be a grammar augmented with actions. $G' \in LALR(k) \Rightarrow G \in LL(k)$ by Corollary 2.2, therefore, any *LALR* (*k*) grammar is *LL* (*k*) if one is free to place actions on the left edge. There exist *LL* (*k*) grammars which are not *LALR* (*k*) such as Grammar G2.3; hence, *LL* (*k*) is strictly stronger than *LALR* (*k*) in the worst case action-placement scenario.
□

Because *LALR* (*k*) is known to be a proper superset of *SLR* (*k*) the relationship between *SLR* (*k*) and *LL* (*k*), follows trivially:

Corollary 2.4: $SLR(k) \subset LL(k)$ *when actions may be placed arbitrarily among grammar productions*.

Proof:
$SLR(k) \subset LALR(k)$ and $LALR(k) \subset LL(k)$, therefore, $SLR(k) \subset LL(k)$ by transitivity when actions may be placed arbitrarily.
□

Previously, *LL* (*k*) was considered weaker than *LR* (*k*) and somewhat weaker than *LALR* (*k*) and *SLR* (*k*) in practice. We have shown that, at least in theory, *LL* (*k*) is as strong as *LR* (*k*) and stronger than *LALR* (*k*) and *SLR* (*k*) when grammars are arbitrarily augmented with actions. It is interesting to note that the relationship between *SLR*, *LALR* and *LR* does not change with arbitrary action placement; i.e. $SLR(k) \subset LALR(k) \subset LR(k)$ which follows directly from Corollaries 2.2, 2.4, and Theorem 2.1.

Augmenting an *LR* grammar with semantic actions can introduce ambiguities because productions must be ''cracked'' to create reductions corresponding to the action positions. This is well known to anyone who has developed translators based on *LALR* (1) grammars. Murphy's Law predicts that the position in a grammar where an action is needed most is precisely the position where rule cracking will introduce an ambiguity. In contrast, *LL* (1) grammars are insensitive to action placement, but many useful grammars are not *LL* (1).

Parsers are currently restricted to a single terminal of lookahead primarily because *LR* (1), upon which most parsers are based, is theoretically equivalent to *LR* (*k*) and because $k > 1$ terminals of lookahead can lead to exponentially large parsers. Algorithms for constructing *LR* (*k*) and *LL* (*k*) parsers can also be significantly more complicated than those for *LR* (1) or *LL* (1) parsers and, are generally, not considered worth the effort. However, $k > 1$ terminals of lookahead are necessary in order to relax the sensitivity of *LR* grammars to action placement and to increase the

recognition strength of *LL* parsers. Although k terminal lookahead presents some challenges in implementation, the difficulties can be overcome as proposed by this thesis. The next section describes how other researchers have attacked the problem of deterministic parsing using $k>1$ terminals of lookahead and contrasts these strategies with our approach.

2.3 Comparison to Previous Work in $LL(k)$ and $LR(k)$ Parsing

$LL(k)$ [RoS70] and $LR(k)$ [Knu65] parsing has been studied vigorously from a theoretical perspective; e.g., the research of [AhU72, DeR71, HSU75, Knu71, LeS68, Sip82, SiS82, SiS83, SiS90, Ukk83]. Practical construction of such parsers and their variants has been largely avoided due to the apparent unavoidable exponentiality in k. Indeed, very few papers actually consider computing lookahead information for $k>1$ ([ADG91, KrM81] are exceptions). Most practical work has centered around the $k=1$ case, e.g., [DeP82, Ive86, Joh78, MKR79, Pen86] and common textbooks such as [ASU86, FiL88].

It has long been known that modulating the lookahead depth according to the needs of each inconsistent parser state can reduce parser size; e.g., [DeR71] suggested this in his paper on $SLR(k)$. Others have taken this one step further to allow infinite regular lookahead languages [BeS86, CuC73] instead of normal k-bounded regular languages. We use varying amounts of lookahead in order to reduce the time required to compute lookahead information as well as to reduce the resulting parser size. [KrM81] does not consider modulating k when computing $LALR(k)$ lookahead sets and indicates that $k=1$ is the practical lookahead depth for their method. [ADG91] defers computation of lookahead sets for $LR(k)$ parsers until parser runtime and uses only as much lookahead as necessary to make each decision. A similar technique can be used statically, which we employ for the variants of $LL(k)$ and $LR(k)$, at grammar analysis time, to compute only that lookahead information necessary to render a state deterministic and for only those states needing lookahead.

Virtually all parsing theory work considers the lookahead k-tuples associated with a decision state to be atomic. On the other hand, we consider individual terminals to be atomic, which results in greater flexibility with regard to grammar analysis and parser construction. We define $LL^1(k)$ and $LR^1(k)$ parsers that examine only individual terminals at the various lookahead depths and no m-tuples ($1 \le m \le k$). These parsers use a covering approximation to full lookahead sets — with potentially linear time and space lookahead computation complexity that reduces lookahead information size from $O(|T|^k)$ to $O(|T| \times k)$. The approximate lookahead sets can be used to construct efficient decisions which handle most parsing decisions. Chapter 7 discusses the generalization of $LL^1(k)$ and $LR^1(k)$ decisions to $LL^m(k)$ and $LR^m(k)$ decisions, which compare subsequences of size at most m (composed of terminals from contiguous or noncontiguous lookahead depths).

To facilitate parsers that use only the necessary information to make state transitions, parsers with heterogeneous states must be constructed. Most parsing work revolves around parsers composed of tables and an interpreter: The parsers have homogeneous states; i.e. each state must be as complex as the most complex needed to parse the particular language of interest. [BeS86, CuC73] build different cyclic-DFA lookahead decisions at each $LR(0)$ nondeterminism, but still employ states that are homogeneous, as each state may have a lookahead DFA. [ADG91] compute lookahead at parse time, rather than at analysis time, by adding a new *LR* action called *look*; again, parser states are homogeneous. [Pen86] developed a parser generator that generates non-interpretive $LALR(1)$ parsers in 8086 assembly language. Later, [Rob90] developed a similar method of encoding non-interpretive bottom-up parsers called ‘‘Recursive Ascent’’ parsers. Both of these authors were concerned with parsing speed. On the other hand, we are concerned with varying the complexity of decision states to reduce the exponential worst-case nature of $LL(k)$- and $LR(k)$-based parsers to near-linear typical behavior.

Algorithms for computing $k>1$ lookahead sets are rare and typically operate on canonical $LR(0)$ machines (e.g., [KrM81]) or the grammar itself. If lookahead depths greater than one are to be used, practical algorithms and data structures for computing lookahead sets, ensuring parser determinism, and parser construction must be developed. We represent grammars as a group of NFA’s, called grammar lookahead automata (GLA’s), which cover the language generated by the CFG. Because lookahead strings form a finite regular language, we consider lookahead information to be acyclic DFA’s, which we encode as child-sibling trees. Lookahead computations can then be elegantly described as a form of constrained traversal of a GLA; the computation is similar to NFA to DFA conversion. To further simplify lookahead computation, we define lookahead operations as $LOOK_k$ rather than as combinations of $FIRST_k$ and $FOLLOW_k$.

The literature discusses the $LL(k)$ and $LR(k)$ classes separately. While the two strategies are very different in terms of parser state construction, they are identical in terms of how they make parser lookahead decisions. The only difference between an $LL(k)$ parser decision state and an $LR(k)$ parser decision state is the type of actions induced by the lookahead information. For example, an $LL(k)$ parser may ‘‘predict’’ a production upon some terminal sequence whereas an $LR(k)$ parser may perform a ‘‘shift’’ or ‘‘reduce’’. We abstract lookahead decisions, regardless of the parsing method, to an **induces** relation that maps lookahead sequences to parser actions. In this manner, many aspects of $LL(k)$ and $LR(k)$-based parsing may be discussed together.

Previous tests for the various grammar properties have complexities that are always exponential in nature and do not compute the lookahead sets needed for parser decision states. Our method computes lookahead sets and compares them to determine if the appropriate property is satisfied. This grammar-driven approach enables us to use minimal lookahead and to compute lookahead only for those decisions that require it. Figure 2.2 summarizes the methods used to test grammars for $LL(k)$ determinism.

Method	Description
Transformation to *LR*	This method transforms the $LL(k)$/$SLL(k)$ determinism problem into the $LR(k)$/$SLR(k)$ determinism problem which can be solved in space $O((k+1)^2 \times \|G\|^2)$ and in time $O((k+1)^3 \times \|T\|^k \times \|G\|^2)$ [HSU75]. Using the results of [Bro74], [HSU75] showed that the *LL* determinism problems were easily reducible to *LR* determinism problems and, hence, they could be solved in the same space and time complexity. These algorithms rely upon the construction of a set of automata denoted $M_{LR(u)}(G)$ (in the notation of [SiS90]), for some CFG G, which accepts those viable prefixes of G which may be followed by $u \in T^k$ in some right sentential form [HSU75].
Dual of *LR*	A dual to the usual *LR* canonical parser exists for *LL* [SiS82, SiS83, SiS90]. Similarly, [Si83, SiS90] present a scheme for $LL(k)$ and $SLL(k)$ testing which dualizes the construction of $M_{LR(u)}(G)$ automata for *LR*, yielding $M_{LL(u)\text{–}set}(G)$. This dual test is more efficient than transforming to *LR* and using the *LR* testing algorithms. Specifically, testing for the $LL(k)$ property is a factor of $\|G\|$ faster than testing for the $LR(k)$ property, but introduces an extra 2^k term: it can be solved in space $O(2^k \times \|G\|)$ and in time $O((k+1) \times 2^k \times \|T\|^k \times \|G\|)$.
Compare lookahead sets	Computes lookahead sets for each alternative production at a decision point and verifies that the lookahead sets have no lookahead sequences in common up to a depth of k. Lookahead sets rarely grow to the upper bound of $O(\|T\|^k)$ in size and rarely need all k terminals; hence, an algorithm that examines only those permutations of T^k which are necessary for each decision is potentially useful in practice. Comparing production lookahead sets has exponentially complex time and space requirements like the other methods, but performs better in practice; e.g., testing for the $SLL(k)$ property is $O((\|G\| + \|P\|^2/\|N\|) \times k \times \|T\|^k)$.

Figure 2.2 Comparison of $LL(k)$ Determinism Methods

The previous methods are obviously impractical as they test multiple automata against permutations of length k sequences of terminals. This is not useful because $|T|^k$ is impractical for even small $|T|$ and k. Table 2.2 demonstrates the impracticality of the previous analysis algorithms. It reflects how much time is required to create $|T|^n$ lookahead n-strings and print them to the null device (`/dev/null`). The times do not include the effort that would be necessary to examine the small automaton, $M_{LL(u)}$, associated with each input permutation u.

Table 2.2 Time to Create $|T|^n$ Lookahead Permutations ($|T|$=100)

create $\|T\|^n$ permutations	lookahead $n \leq 4$			
	1	2	3	4
time (secs)	0.0	0.9	146.2	53564.3

Computing and then comparing lookahead sets has an additional advantage over the theoretical methods: Productions may easily be tagged with the lookahead sequences which render the associated decision nondeterministic. Very specific warning messages may be reported.

Thus far, we have provided motivation for the use of $k>1$ terminals of lookahead and described how others have attacked the problem of constructing $LL(k)$ and $LR(k)$ parsers. The next section provides an impractical, but useful way to view parsing. Specifically, we define an optimal parser which examines each input symbol at most once; in order to reduce the number of lookahead inspections, an optimal parser construction algorithms would have to consider individual terminals rather k-tuples and would construct parsers that make decisions of varying complexity. Using this fundamental change of perspective, Chapters 3 and 4 develop methods for constructing practical $LL(k)$- and $LR(k)$-based parsers.

2.4 Optimal Parsing of Deterministic Languages

In theory, parsers change state by examining a k-tuple and then shifting the input by at most one symbol (terminal), which implies that tokens will be tested at least k times. $LL(k)$ and $LR(k)$ parsing is widely held to be time $O(n)$ for an input of size n; however, the time complexity is more precisely $O(k \times n)$. The ability to construct parsers that examine each input symbol exactly once, regardless of k, has little impact on parsing speed as n is normally much larger than k; however, this notion emphasizes the fact that parser decisions can reduce their time and space complexity by using less than the maximum lookahead depth and/or by ignoring some lookahead depths.

Our approach to parsing for $k>1$ relies upon the idea that individual terminals, rather than k-tuples, are the basic, atomic, entity of parsing. Optimal parsing can be seen as motivating the ‘‘splitting’’ of the k-tuple atom. This new definition that a terminal comparison is an atomic operation directly motivates the linear approximations $LL^1(k)$ and $LR^1(k)$ which compare 1-tuples (sets) rather than k-tuples; see Section 3.7.1 for a precise definition. Further, we denote parsers or parsing decisions, whose largest atomic operation is an $m-$tuple comparison, $LL^m(k)$ and $LR^m(k)$; see Section 7.24.

Parsers normally examine lookahead to make a decision and then promptly ‘‘throw out’’ much of the information thus obtained. An optimal parser must record the result of decisions by state splitting — i.e. split states that have at least one edge that is traversable by more than one lookahead sequence and then remove impossible items from each of the new states. The new states/edges indicate that a certain lookahead configuration was seen and, hence, the set of possible future actions is smaller for each of them. The number of new states replicated for a given state (for both $LR(k)$ and $LL(k)$) will be roughly equal to the number of possible unique input sequences that induce a transition to that state. Both *LR* and *LL* parsers may employ this scheme even though they record lookahead configuration information differently.

This section demonstrates state splitting for $LL(k)$ and $LR(k)$-based parsers and, for $LL(k)$ grammars, defines an Optimal Normal Form (ONF) which yields optimal parsers using normal construction techniques.

2.4.1 Structure of Parser Comparison Sequences

Conventional parsing examines each input symbol k times. However, an optimal parser inspects each input symbol at most one time regardless of the size of the lookahead buffer, k. This section illustrates the structure of lookahead decision sequences for conventional parsers, optimal parsers and hybrid parsers that use a variety of decision templates.

Figure 2.3 illustrates the usual parsing strategy, for $k=3$, of comparing 3-tuples, shifting the input by one, and then comparing another 3-tuple; $\delta_i(\tau_1, \tau_2, \tau_3)$ is the i^{th} 3-tuple decision, based upon the lookahead buffer (τ_1, τ_2, τ_3), and a_i is the i^{th} input symbol (token).

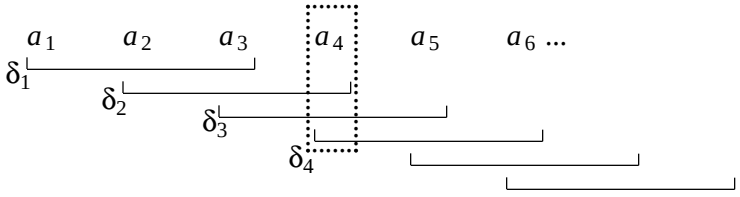


Figure 2.3 Conventional Parsing for *k*=3

The dotted box associated with decisions δ_2 ... δ_4 highlights that after the first *k* tokens have been read, each token, τ_i, will be examined *k* times. In contrast, an optimal parser might perform the sequence of comparisons illustrated in Figure 2.4.

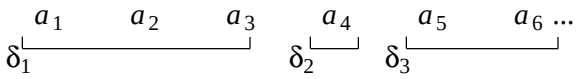


Figure 2.4 Optimal Parsing for *k*=3

Many decisions can also be made from contextual information alone; i.e. no lookahead is needed at all. Therefore, an optimal parser may actually use fewer than one comparison per input token, but generally, the number of comparisons depends on the parsing method.

If one could take advantage of the fact that the grammar had few constructs that required three tokens of lookahead, one could make comparison sequences similar to that in Figure 2.5.

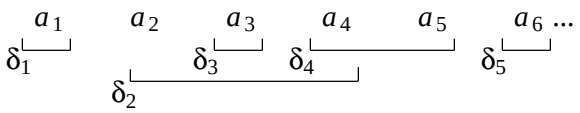


Figure 2.5 Near-Optimal Parsing where *k*-tuple Is Not Atomic

Occasionally, a decision will require three tokens, but in general decisions are made using only one token (or none at all).

2.4.2 Optimal $LL(k)$ Parsing

$LL(k)$ parsers have only type of action that need be induced by lookahead: Before recognition of a nonterminal begins, one of the alternative productions must be predicted using up to k terminals of lookahead. If less than k symbols are consumed before the lookahead buffer must be examined again, the parse is not optimal as input symbols will be examined more than once. This section defines an optimal normal form (ONF) for $LL(k)$ grammars for which a parser, built in the conventional manner, is optimal. In addition, this section outlines a method for splitting the states of an $LL(k)$ parser to render it optimal.

A grammar comprised of productions that always have k terminals on the left edge will result in an optimal $LL(k)$ parser; such a grammar is said to be $LL(k)$ optimal. Although many definitions are possible, we define an optimal $LL(k)$ grammar as follows:

Definition: *A grammar is said to be in* Optimal $LL(k)$ Normal Form, *denoted ONF, if each nonterminal is of the form:*

$$\begin{aligned} A &\rightarrow \alpha_1\beta_1 \\ A &\rightarrow \alpha_2\beta_2 \\ &\ldots \\ A &\rightarrow \alpha_m\beta_m \end{aligned}$$

where $\alpha_i \in T^k$, $\beta_i \in (N \bigcup T)^*$, m is the number of productions of A, and $\alpha_i \neq \alpha_j$ for $i \neq j$; i.e. all productions may be predicted unambiguously and without examining tokens needed for the prediction of another nonterminal's productions. Our definition of ONF is similar to Greibach Normal Form [Gre65] except that we restrict our discussion to $LL(k)$ grammars, require k token sequences on the left edge (versus one), and do not restrict the form of productions past the k-sequence prefixes.

The simplest grammar in ONF is of the form:

$$A \rightarrow \alpha$$

which has no decision to make as there is a single production and, hence, a single choice; nonterminal A is $LL(0)$. In contrast, the $LL(2)$ Grammar G2.6 has two decision points (on the left edges of A and B):

$$\begin{aligned} A &\rightarrow ab \\ A &\rightarrow aB \\ B &\rightarrow c \\ B &\rightarrow d \end{aligned} \qquad \text{G2.6}$$

For nonterminal A, lookahead set (a,c), (a,d) induces a *predict* $A \rightarrow aB$ action. The τ_2 lookahead components c and d used to predict A's productions will be matched in B, which must also be used to distinguish between B's alternative productions. Hence, this grammar is not in an

optimal form as at least one input symbol will be inspected more than once. In general, examining an input symbol in one decision excludes its use by any other decision. Figure 2.6 presents the *LL* (2) machine. Most transition arcs are labeled as *a* where *a* is the input to be consumed during transition; transitions out of production prediction states consume no input and are of the form $\varepsilon \mid v$ where *v* is the lookahead component which must be present to make the transition. Calls to other nonterminals are as edges labeled with the nonterminal name (upper-case Latin letter). Some states are labeled for reference in the text.

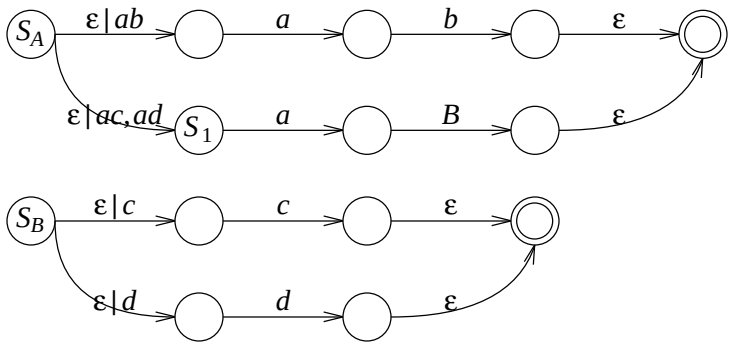


Figure 2.6 *LL* (2) Machine for Grammar G2.6

To make a transition out of state S_A, we examine (τ_1, τ_2). Entering state S_1 by traversing edge $\varepsilon \mid ac, ad$ does not record which of *ac* or *ad* was found on the input stream. Later, in state S_B, the *c* or *d* must again be used to distinguish between productions. This is the source of Grammar G2.6's nonoptimality. If one were to split state S_1 into two new equivalent states, the results of the prediction in state S_A could be ''remembered''. Figure 2.7 shows the results of splitting state S_1.

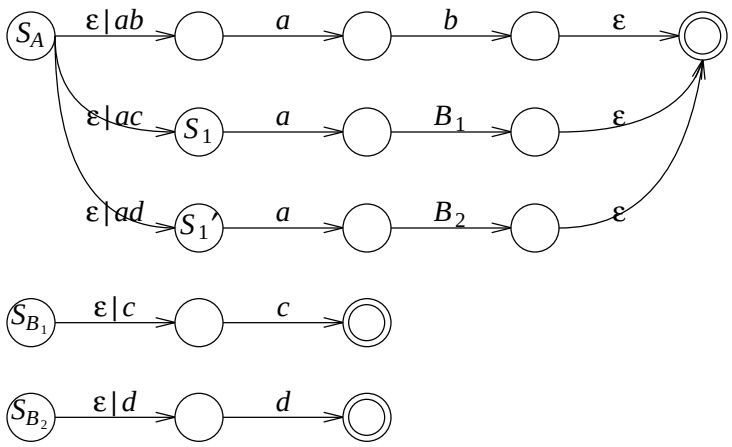


Figure 2.7 Partial Optimal *LL* (2) Machine for Grammar G2.6

Splitting state S_1 into S_1 and S_1' affects the states reachable from S_1 and S_1' (duplicates and simplifies) because more information is available than before as to what is coming on the input stream. For example, state S_{B_1} no longer has a decision to make because the previously present states for $B \rightarrow d$ cannot possibly be visited. Similarly, state S_1' has only one transition as $B \rightarrow c$ cannot possibly be applied.

Splitting the decision state S_1 to ''remember'' the results of examining τ_1 and τ_2, corresponds to duplicating $A \rightarrow aB$ and separating the productions of *B*. Separating *B*'s productions removes a decision because the correct production to match will be determined by the invoking nonterminal, *A*. Figures 2.6 is to Figure 2.7 as Grammar G2.6 is to Grammar G2.7.

$$\begin{aligned} A &\rightarrow ab \\ A &\rightarrow aB_1 \\ A &\rightarrow aB_2 \\ B_1 &\rightarrow c \\ B_2 &\rightarrow d \end{aligned} \qquad \text{G2.7}$$

Parser state splitting is analogous to the instantiation of nonterminals in the grammar domain. For example, Grammar G2.8 is the same as Grammar G2.6 except that the productions of *B* have been instantiated into the reference to *B* in nonterminal *A*, which yields an optimal grammar.

$$\begin{aligned} A &\rightarrow ab \\ A &\rightarrow ac \\ A &\rightarrow ad \end{aligned} \qquad \text{G2.8}$$

Occasionally, instantiation will force left-factoring, which is not always possible when grammars are augmented with semantic actions. For example, consider Grammar 2.9 which represents an *LL* (1) grammar after instantiation of some nonterminal into *A*. It must be left-factored to remain

LL (1).

$$\begin{aligned} A &\rightarrow ab \\ A &\rightarrow b \bullet_1 c \\ A &\rightarrow b \bullet_2 d \end{aligned} \qquad \text{G2.9}$$

After left-factoring, we obtain Grammar G2.10.

$$\begin{aligned} A &\rightarrow ab \\ A &\rightarrow bB \\ B &\rightarrow c \\ B &\rightarrow d \end{aligned} \qquad \text{G2.10}$$

If actions had been placed at positions $\bullet_1$ and $\bullet_2$, however, this left-factoring could not take place as actions cannot be merged. Lookahead decisions could be attached to actions to remember their original context, but optimal parsers are intractable anyway; hence, we are not overly concerned by this issue.

It is not always possible to generate optimal grammars by instantiation. Consider Grammar G2.11.

$$\begin{aligned} S &\rightarrow AB\$ \\ A &\rightarrow ab \\ A &\rightarrow a \\ B &\rightarrow c \\ B &\rightarrow d \end{aligned} \qquad \text{G2.11}$$

Nonterminal *A* is nonoptimal as two symbols are required to predict the productions and the second symbol is also required to predict the productions of *B* after the recognition of *A*. State splitting can be done in several ways here, but one grammar transformation results in Grammar G2.12.

$$\begin{aligned} S &\rightarrow A\$ \\ A &\rightarrow abB \\ A &\rightarrow aB_1 \\ A &\rightarrow aB_2 \\ B &\rightarrow c \\ B &\rightarrow d \\ B_1 &\rightarrow c \\ B_2 &\rightarrow d \end{aligned} \qquad \text{G2.12}$$

In this case, we have moved what follows the reference to *A* to the ends of the productions of *A*, without changing the language. Then, instantiation proceeds as before. The reference to *B* in $A \rightarrow abB$ is not instantiated for two reasons: First, it would cause Grammar G2.12 to become non-*LL* (2) and, second, it is unnecessary to do so as there is a prefix of length 2. Another type of state splitting can be done which records the result of the lookahead decision in *B* by returning to

a different state in A for each lookahead sequence.

This section defined an optimal normal form, ONF, for $LL(k)$ grammars for which a parser, built in the normal fashion, is optimal. Further, a scheme for splitting parser states was outlined and was shown to be analogous to a grammar transformation. The next section characterizes when $LR(k)$ parsers need to split states to become optimal and provides an example state splitting strategy.

2.4.3 Optimal $LR(k)$ Parsers

The optimal $LL(k)$ parsers of the previous section made decisions in the states associated with the left edge of productions. $LR(k)$ parsers, on the other hand, make lookahead decisions in any state that is $LR(0)$ inconsistent; i.e. any state with more than one item in the core that contains a reduce item, $A \rightarrow \alpha \bullet$. Optimal $LR(k)$ parsers split states in a manner similar to optimal $LL(k)$ parsers. This section characterizes $LR(k)$ parser state splitting via an example grammar for which the $LR(2)$ machine and optimal $LR(2)$ machine are given.

Optimal $LR(k)$ parsers have exactly one state transition arc per lookahead sequence, which requires that target states be split. In this way, the results of each lookahead examination, performed in some state p, are recorded for use by states reachable from p. Figure 2.8 shows a generic decision state.

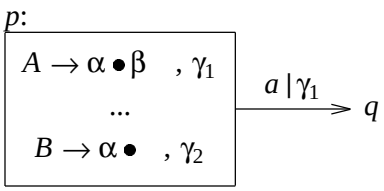


Figure 2.8 Generic $LR(k)$ Parser Decision State

where $\beta = x\beta'$ and $x \in V$. Again, transitions are marked with $\alpha | \beta$ where α is the input terminal to consume and β is the lookahead component that must be on the input stream to make the transition. If $|\gamma_1| > 1$ then states reachable by that transition may be nonoptimal. Therefore, the transition γ_1 out of state p must be separated into transitions with only one lookahead sequence, which forces duplication of q — one for each element of γ_1.

To better illustrate optimal $LR(k)$, consider Grammar G2.13, which is $LR(2)$.

$$
\begin{array}{l}
S \rightarrow Aa\,\$\,\$ \\
A \rightarrow aBc \\
A \rightarrow \\
B \rightarrow b \\
B \rightarrow
\end{array}
\qquad \text{G2.13}
$$

A partial $LR(2)$ machine, in the notation of [ASU86] except for the $u \mid v$ edge notation, is given in Figure 2.9.

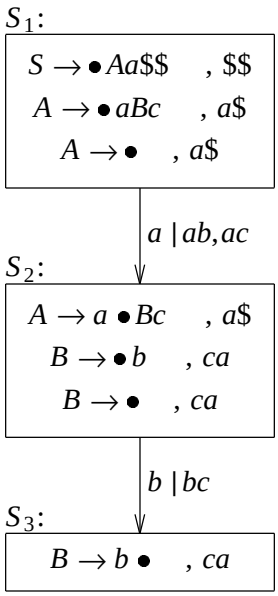


Figure 2.9 Partial $LR(2)$ Machine for Grammar G2.13

In state S_1, input a induces a *shift* for $A \rightarrow \bullet aBc$ and a *reduce* for $A \rightarrow \bullet$. Hence, a lookahead depth of two is required — input ab and ac induce a *shift* for $A \rightarrow \bullet aBc$ and input $a\$$ induces a *reduce* for $A \rightarrow \bullet$. The second input symbol, one of $\{b,c,\$\}$, will be used again to induce a *shift* for $B \rightarrow \bullet b$ or a *reduce* for $B \rightarrow \bullet$; Grammar G2.13 is nonoptimal.

The $LR(2)$ machine in Figure 2.9 can be made optimal by splitting S_2 and then duplicating/simplifying all states reachable from S_2. Such an optimal machine is given in Figure 2.10.

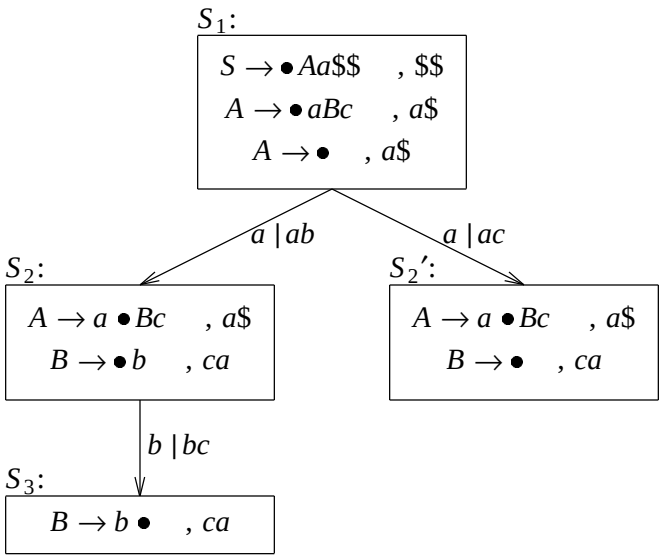


Figure 2.10 Partial Optimal $LR(2)$ Machine for Grammar G2.13

States S_2 and S_2' are simplified because the transitions entering them have only a single lookahead sequence; e.g. the previous item $B \rightarrow \bullet$ can no longer be reached because input sequence *ac* will no longer force a transition to state S_2. State S_3 is duplicated, but simplification removes all its items and, therefore, it disappears.

Optimal $LR(k)$ and $LL(k)$ parsers are very similar. Both split states (duplicate subgraphs) to ''remember'' the terminal sequence matched on the input stream. After splitting, transitions will have only one terminal sequence in the lookahead component of the transition label. Because of these simplified transition arcs, the states in the duplicated subgraphs will be much simpler as the set of possible items will be greatly diminished. The only real difference between optimal $LL(k)$ and $LR(k)$ parsers is that, generally, a parser transformation has a corresponding grammar transformation for $LL(k)$ parsers. With regards to error detection, optimal parsing detect errors exactly as early as before except that the recognition may occur in a different state due to state splitting. Splitting states increases the context information available to a parser and, hence, will not delay error detection. Normally, error detection is delayed only by inaccurate lookahead routing information. For example, $SLL(k)$ parsers have less accurate lookahead information than $LALL(k)$ parsers; the effect is that $SLL(k)$ parsers do not detect errors as quickly as $LALL(k)$ parsers even though both parsers have the same number of states.

The method of obtaining optimal parsers described in this section does not consider user-defined semantic actions. A more general approach would attempt to liberate the actual lookahead examinations from the decision states in an effort to avoid left-factoring, which cannot be generally done in a grammar augmented with actions, and explosive state splitting. We did not explore this issue due to optimal parsings fundamental lack of practicality.

In this chapter, we gave motivation for the use of $k > 1$ terminals of lookahead and provided a comparison of our approach to the previous work done in the area of $LL(k)$ and $LR(k)$ parsing. Our strategy revolves around parsers that make decisions of varying complexity and parser decisions that consider terminals, rather than k-tuples, atomic entities; optimal parsing can be seen as inspiring this strategy. The next two chapters use the observations given in this chapter to provide a framework for grammar representation, parser state construction, decision state abstraction, lookahead computation, and lookahead representation.

CHAPTER 3 PARSING

The previous chapter described why more than a single lookahead symbol is needed and outlined why the work done previously, in the area of $LL(k)$ and $LR(k)$ parsing, is mostly impractical. To provide a practical means by which parsers with large lookahead buffers may be constructed, we must dispense with the norm of homogeneous parser states, uniform lookahead depth, and atomic k-tuples as in Section 2.4 on optimal parsing. This necessitates a nontrivial change of perspective with regards to parser construction. Hence, this chapter presents new ways to think about representing grammars, parsers, and parsing decisions.

For the purpose of grammar analysis, we store grammars as a group of intertwined NFA's, called grammar lookahead automata (GLA's), that represent the collection of all possible lookahead languages for all grammar positions. In fact, GLA's realize a covering, regular approximation to the underlying CFL of the grammar. The lookahead sequences for a particular parsing strategy of depth k for a position in the grammar correspond to a subset of the sequences of non-ε edges along the walks of length k starting from the associated GLA state. Lookahead computations are described as bounded walks of a GLA where the lookahead sequences are encoded as lookahead DFA; in practice, we represent the lookahead DFA as child-sibling trees. The reader will notice a similarity to NFA to DFA conversion.

Different decision states may have different lookahead requirements. Section 3.7.2 provides empirical data that suggests that the vast majority of lookahead decisions, 98.57% in our study of 22 $SLL(k)$ grammars, can be handled with no lookahead or with a single lookahead symbol. When more than a single token of lookahead is required, it is often the case that a linear approximation to the normal, exponential lookahead information can be used. We define $C^1(k)$ decisions as decisions that look no more than k terminals into the future and compare at most 1-tuples (sets of terminals). To facilitate parsers that take advantage of these decision states of varying complexity, we describe a mechanism for representing parsers with heterogeneous states in Section 3.6.

The various parsing strategies construct parsers with different states and state actions, but the lookahead decisions within decision states are identical. We abstract the concept of a lookahead-to-action mapping as a mathematical relation called **induces** in Section 3.7.4, which isolates the required lookahead information (and it computation) from the details of implementing the mapping. Thus, any transformation or compression on an **induces** relation is applicable to any parsing strategy.

In general, this chapter describes the foundations upon which practical $LL(k)$- and $LR(k)$-based parsers are built. We describe the representation of grammars, heterogeneous parsers, and parsing decisions.

3.1 Grammar Representation

Grammar analysis computes lookahead sets for parser construction and for testing parser determinism. Many algorithms operate on the grammars themselves (stored as simple lists of productions) while some of the *LR*-based algorithms operate on the canonical *LR* machines. We represent grammars, regardless of the parsing strategy, as a collection of NFA's called grammar lookahead automata (GLA's), which realize a covering, regular approximation to the underlying language of the grammar. GLA's represent the collection of all possible lookahead languages for all grammar positions. The possible lookahead sequences of depth k for a position in the grammar correspond to the sequence of non-ε edges along the walks of length k starting from the associated GLA state. The edges found along the walks can be recorded as deterministic finite automata (DFA's), which we store as child-sibling trees. Consequently, all lookahead computations for any $LL(k)$ or $LR(k)$ variant can be elegantly described as constrained walks of GLA's.

Our GLA's are reminiscent of the transition diagrams of [ASU86] except that only terminals and ε may appear as transition (arc) labels. In addition, each nonterminal, A, has ε-transitions emanating from its accept state which point the nodes immediately following references to A. There exists a state in the GLA for each position in the grammar and a transition for each terminal and nonterminal reference appearing in any production's right-hand-side. To construct GLA's, we create a state, p_i for each position (item) in the grammar and two states, p_A and q_A for each nonterminal A where p_A is the nonterminal entry and q_A is the nonterminal accept state; then, we construct GLA transition arcs as per the algorithm in Figure 3.1.

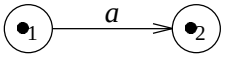
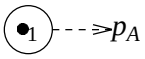
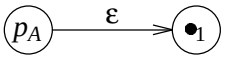
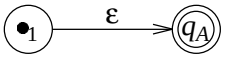
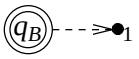
$A \rightarrow \alpha \bullet_1 a \bullet_2 \beta$	Connect the states created for the positions $\bullet_1$ and $\bullet_2$ with a transition labeled a.
	
$A \rightarrow \alpha \bullet_1 A \bullet_2 \beta$	Make an ε-transition from the state created for $\bullet_1$ to state p_A.
	
$A \rightarrow \bullet_1 \alpha$	Make an ε-transition from p_A to the state created for $\bullet_1$.
	
$A \rightarrow \alpha \bullet_1$	Make an ε-transition from the state created for $\bullet_1$ to q_A.
	
$A \rightarrow \alpha B \bullet_1 \beta$	Make an ε-transition from q_B to the state created for $\bullet_1$.
	

Figure 3.1 GLA Construction from CFG

Dashed edges reflect ‘‘pointers’’ to other states; they are not drawn to increase clarity as their inclusion would cause a spagetti effect. This representation of a grammar, G, effectively constructs a single NFA whose envelope covers the underlying context-free language, $L(G)$. It should be stressed that GFA’s are not purely NFA’s because a grammar-to-GLA-state mapping is required.

The language of a GLA is generally larger than the underlying context-free language even when $L(G)$ is regular and it will be up to the algorithms themselves to determine which transitions to follow in the GLA when collecting lookahead information. The sophistication of the algorithm and, hence, the accuracy of this information will often be the distinguishing factor between parser classes; e.g., $SLR(k)$ and $LALR(k)$ differ only in lookahead information [ASU86].

To illustrate the GLA construction algorithm, consider Grammar G3.1.

$$\begin{aligned} A &\rightarrow aAa \\ A &\rightarrow b \end{aligned} \qquad \text{G3.1}$$

The associated GLA is shown in Figure 3.2.

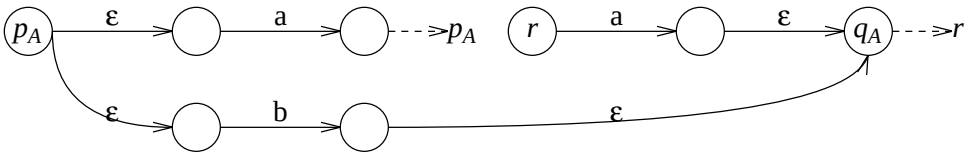


Figure 3.2 GLA for Grammar 3.1

The language described by nonterminal A is $L_1 = \{a^n b a^n \mid n \geq 0\}$ whereas the regular envelope of the GLA is $L_2 = \{a^* b a^*\}$; hence, L_2 covers L_1, $L_1 \subseteq L_2$.

For the most part, GLA’s will be illustrated as if they were constructed via the algorithm in Figure 3.1, however, this representation is idealized for discussion; in practice, slightly different GLA’s are actually constructed. In an effort to reduce grammar-analysis algorithm implementation complexity, a convention concerning GLA states is followed — each state has at most two arcs emanating from it, denoted $p \rightarrow edge_1$ and $p \rightarrow edge_2$, where $p \rightarrow edge_2$ is always labeled ε, if it exists. Figure 3.3 shows how idealized GLA’s differs from the GLA’s actually used.

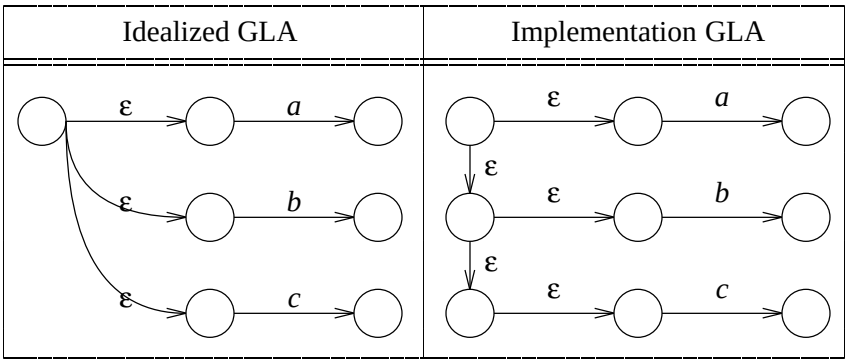


Figure 3.3 Idealized GLA versus Implementation GLA

In addition, because grammars are normally $\$^k$-augmented before analysis ($S' \rightarrow S\$^k$), a reflexive \$-transition to the GLA accept state of the start symbol such as that in Figure 3.4 is constructed.

Figure 3.4 $\$^k$-augmentation of GLA's

Section 4.9 describes lookahead operations in more detail and Section 4.8 describes the representation of lookahead information itself. The next section describes how parsers may be represented using automata with heterogeneous states.

3.2 Heterogeneous Automata in Deterministic Parsing

The previous section described how grammars may be represented in such a way that lookahead computations are conveniently defined as GLA walks; it did not discuss how parsers could be built from the grammars. While most parsers described in the literature employ parsers with one type of parser state, this section describes how parsers may be constructed using automata with different state types.

Parsers with large lookahead buffers that employ *homogeneous automata* (automata with exactly one type of state) are impractical; they are unable to take advantage of the fact that not all states require lookahead and, of the states that do, most require only a single token. To facilitate the construction of parsers as championed by this thesis, a mechanism, by which automata with many different state types can be described, is necessary — mere tables are insufficient.

A *heterogeneous automaton* is an automaton for which each state may perform a different function; specifically, we are interested in states that make transitions by examining different amounts of lookahead and by examining lookahead in different ways. The class of heterogeneous automata includes recursive-descent parsers, heterogeneous *LL*-machines, and heterogeneous *LR*-machines. Recursive-descent parsers typically have a function for each grammar nonterminal and may naturally perform different operations at each decision point. In this section, we describe heterogeneous *LL* and *LR* machines, which are exactly the same except for the actions induced by lookahead examinations; Chapters 5 and 6 describe recursive-descent parsers in more detail. Heterogeneous automaton states consist of a state label, a set of lookahead inspection/action pairs, and a set of state actions; otherwise, these machines have the normal set of terminal and vocabulary symbols, transition mapping, collection of states, accept states, and start state. No implicit ordering for the lookahead inspections is specified, but the state actions are executed only if none of the lookahead actions are executed and in the order specified. For example, consider the template shown in Figure 3.5.

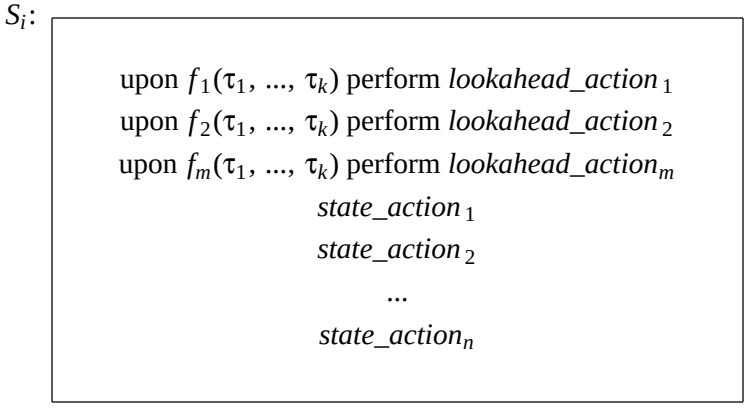


Figure 3.5 Heterogeneous Automaton Template

The lookahead inspection functions, f_i's, have no implicit order so that no restrictions are placed upon the mapping from lookahead to transition action; they may also be functions of the current parser state. The *state_action*$_i$'s are not restricted in any manner (they may have local, global or no effect) and are executed in the order specified.

Constructing a parser with heterogeneous states is essentially the same as for normal recursive-descent, *LL*-machines or *LR*-machines. The sole difference lies in the focus of this thesis — the realization of state transition functions for states that require lookahead. Transitions can also occur without the need for lookahead. For example, *LR*-machines can change state based upon the current stack-top symbol and *LL*-machines can change state without examining anything (assuming valid input). Transitions that do not examine lookahead are simply state actions.

The remainder of this section presents two examples that illustrate the construction of heterogeneous *LL*- and *LR*-machines; the examples use different grammars because *LL* and *LR* parsers generally need lookahead in different states. We begin with the implementation of an *LL*-machine for Grammar G3.2.

$$
\begin{aligned}
&A \rightarrow B \\
&A \rightarrow C \\
&B \rightarrow ab \\
&B \rightarrow cd \\
&C \rightarrow ae
\end{aligned}
\qquad \text{G3.2}
$$

Grammar 3.2 has two nonterminals with more than one alternative production and, hence, two decision states exist in the *LL*-machine. Predicting alternatives of *A* requires a lookahead depth of two, predicting alternatives of *B* requires a lookahead depth of one and *C* has no lookahead requirements as there is only one alternative. The normal *LL*(2)-machine, akin to [ASU86], is depicted in 3.6.

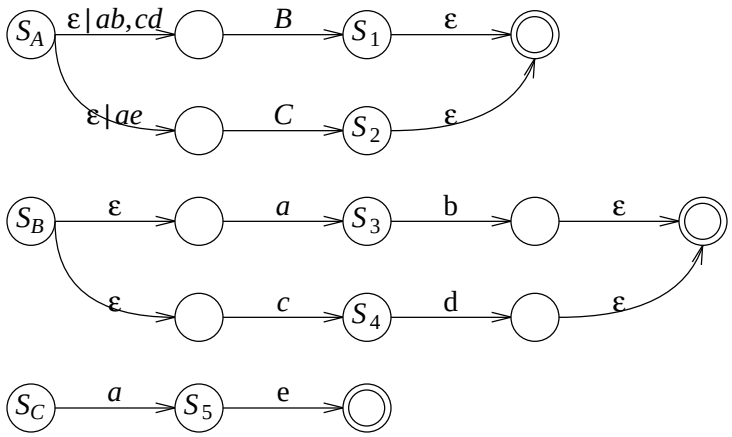


Figure 3.6 *LL*(2)-machine for Grammar 3.2

We have augmented the notation of [ASU86] to include the lookahead components on transition arcs; i.e. ε | *v* where *v* is the lookahead component. The diagrams in [ASU86] are *LL*(1) and, hence, the lookahead components are obvious whereas, here for $k > 1$ they are not. Using a heterogeneous state mechanism, states S_A, S_B, and S_C could be represented by the states in Figure 3.7.

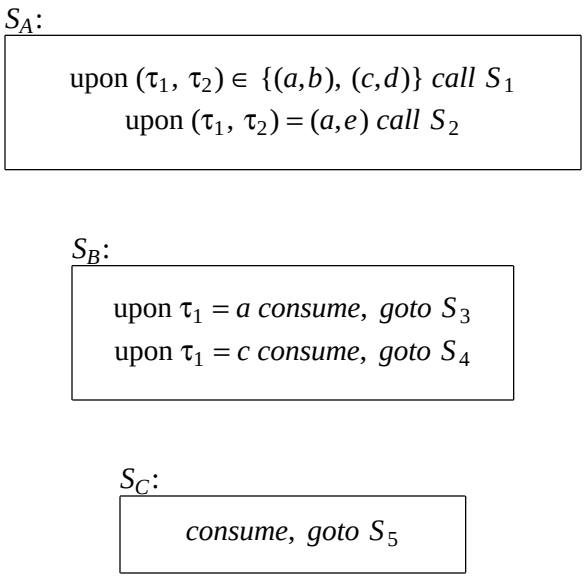


Figure 3.7 Heterogeneous Automaton States S_A, S_B, and S_C

States S_B and S_C would be unnecessarily complex if a homogeneous automaton were used because each state is as complex as the state with the most complicated decision. Note that, since nonterminal *C* has no decision to make, it may merge all states into one of the form depicted in Figure 3.8 (assuming valid input). State S_A can be reduced so that it only examines the second token of lookahead, τ_2, because it uniquely predicts which production to apply; the first token of lookahead has no routing information since *a* is a common prefix.

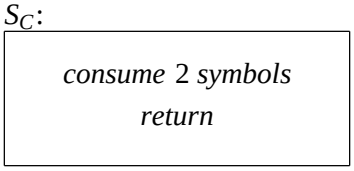


Figure 3.8 Compression of States for Nonterminal *C*

Just as *LL* parsers can be described using heterogeneous automata, *LR* parsers can be expressed using the heterogeneous automata state template. Consider Grammar G3.3, which better illustrates *LR*'s different state requirements.

$$S \rightarrow A\$\$$$
$$A \rightarrow Bab$$
$$A \rightarrow ac$$
$$B \rightarrow$$

G3.3

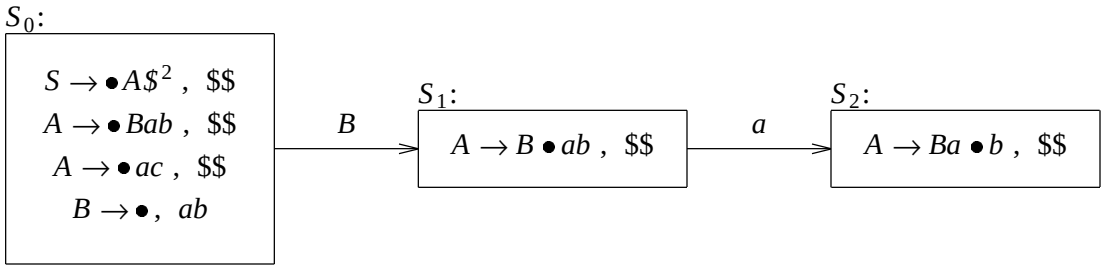


Figure 3.9 Partial *LR*(2)-machine for Grammar 3.3

A portion of the *LR*(2)-machine is shown in Figure 3.9. State S_0 requires a lookahead depth of two to resolve the *shift*/*reduce* conflict between $A \rightarrow ac$ and $B \rightarrow$. However, states S_1 and S_2 do not need to examine lookahead at all to effect a state switch — the transitions are a function of the current state. Using the heterogeneous automata state templates, this partial *LR*(2)-machine can be efficiently encoded as shown in Figure 3.10.

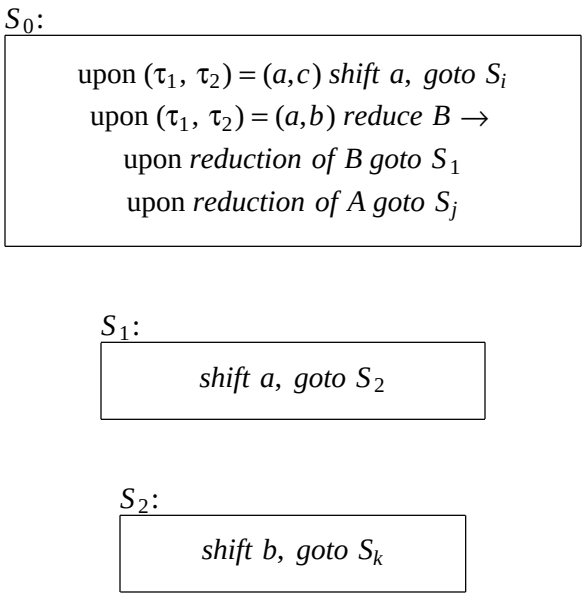


Figure 3.10 Heterogeneous Automaton States S_0, S_1, and S_2

where S_i and S_j go to portions of the machine not shown. As before, state compression can be performed; i.e. states S_1 and S_2 could be merged easily into a single state that shifts *ab* and goes to S_k. *LR* parser compression has been studied in detail (e.g. [AhU73, DeM75, LaL76]), but this very effective type of compression is uncommon because it is possible only when heterogeneous states are considered.

Heterogeneous automata for both *LL* and *LR* parsers have been illustrated in this section. A striking similarity exists between the machines for the two parsing strategies, which emphasizes the fact that the states and the number of states may differ, but lookahead decisions are simply mathematical relations. The next section explores this notion in detail.

3.3 Parsing Decisions

Parser construction is composed of three tasks: First, lookahead information must be computed. Second, parser decision states must be built using this lookahead information. Thirdly, the individual lookahead decision states are examined to ensure determinism. Convention wisdom has it that the tasks are impractical due to exponentially large lookahead requirements. While the worst-case will always be exponential, the common case can be reduced to near-linear performance; Section 3.7.1 describes a strictly linear decision type called $C^1(k)$ that can often be used as an approximation to full $C(k)$.

This section characterizes when parsers need to examine lookahead and then provides a mathematical relation, called **induces**, that abstracts parser decision state transitions and isolates the required lookahead information from the details of implementing a mapping. In doing so, we argue that $LL(k)$ and $LR(k)$ parsers are identical from a lookahead decision point of view.

3.3.1 $C^1(k)$ Decisions

$C(k)$ parser decision states typically have been considered a simple matter — k-tuples are examined and the appropriate parser action is induced. The lookahead information being examined is exponentially large in the worst case and, thus, these decision states were explored from a theoretical point of view. The work on *LR*-Regular languages [CuC73, BeS86] allows unbounded regular lookahead languages instead of normal k-bounded regular languages to induce parser action. But, these lookahead languages are recognized by DFA that are also exponentially large. Just as the lookahead information for $SLR(k)$ parsers is an inaccurate superset of the $LALR(k)$ information ($SLR(k)$ parsers use context-independent sets and $LALR(k)$ use context-dependent sets [SiS90]), another type of covering approximation to lookahead information can be defined that simplifies lookahead computation and reduces the size of decisions states to a linear function of k.

We introduce $C^1(k)$ decisions as decisions that examine at most k future terminals and use only 1-tuple (set) comparisons to induce parser actions. The lookahead information for $C^1(k)$ can then be compressed to k sets of terminals, $O(|T| \times k)$, rather than the normal $O(|T|^k)$ k-tuples. The i^{th} set in the $C^1(k)$ information is the collection of all terminals visible at depth i starting from a grammar position where the definition of ‘‘visible’’ depends on the parsing strategy; $C^1(1)$ is equivalent to $C(1)$ because $C(1)$ is also the set of terminals that can be matched next — one terminal in the future. The definition of this approximate lookahead can be taken advantage of when computing lookahead information and when constructing parser decision states. Computing $C^1(k)$ lookahead information is not forced into exponentiality by the size of the information as is computing $C(k)$ information. In the same sense, decision states are no longer exponentially large because the approximate lookahead information is linear in size.

The $C^1(k)$ space reduction comes at the cost of inaccuracy. For example, the two lookahead tuples (a,b) and (c,d) that induce some parser action have $C^1(2)$ lookahead set sequence $\{a,c\},\{b,d\}$. The $C^1(2)$ decision strategy, however, would map any tuple with $\{a,c\}$ at lookahead depth one and $\{b,d\}$ at lookahead depth two to that parser action — 2-tuples (a,b), (a,d), (c,b), and (c,d).

In general, all of the $LL(k)$ and $LR(k)$ variants can compute the approximate lookahead as a quick first attempt to resolve $C(0)$ nondeterminisms. Unfortunately, full $LL(k)$ and $LR(k)$ cannot use this approximate lookahead during grammar analysis, but they may arrive at the same information (the hard way) by compressing the full lookahead information. Any decision state may

take advantage of the $C^1(k)$ information; Chapter 5 examines $SLL^1(k)$ and Chapter 7 describes how the other deterministic strategies may employ $C^1(k)$ information and decision templates.

When $C^1(k)$ decisions cannot be used in place of the full $C(k)$ decisions (which are exponential in k), it is important to reduce k to the minimum possible. The next section describes the lookahead depths required for parser decisions with $SLL(k)$ used as a vehicle for exploration.

3.3.2 $SLL(k)$ Lookahead Characteristics

Most parser transitions require no lookahead — state information alone is sufficient to determine a course of action. Even when lookahead is required to distinguish between state-transition arcs, usually, only a single lookahead terminal is required. It is precisely this fact that brings $LL(k)$- and $LR(k)$-based parsers for $k>1$ into the realm of practicality. Due to the exponential state explosion for full $LL(k)$ and $LR(k)$, we focus on the linearly-sized variants. In particular, we choose $SLL(k)$ parsers as a practical, easy to construct alternative that demonstrates out approach to parsing with large lookahead buffers.

This section presents statistics about the nature of $SLL(k)$ parsing decisions that involve lookahead. 22 sample grammars, submitted by PCCTS [PDC92] users, are analyzed for lookahead requirements; see the Appendix for a description of the grammars. There are nondeterministic decisions in most of the grammars, although these decisions are usually resolved correctly during parser construction; e.g., the dangling-else-clause construct is non-$LL(k)$, but is handled correctly by matching the ‘‘else’’ as soon as possible.

Table 3.1 demonstrates that the vast majority of $SLL(k)$ decisions, 98.57%, are $SLL(0)$ or $SLL(1)$. A quick calculation indicates that, of the decisions that require lookahead, 98.81% require only a single lookahead symbol.

Table 3.1 Lookahead Requirements for 22 Sample Grammars

grammar	$\|T\|$	$\|N\|$	decisions	lookahead $n \leq 3$				non-*SLL*(3)
				0	1	2	3	
S1	81	113	311	198(63.6%)	107(34.4%)	3(0.9%)	0	3
S2	66	52	150	98(65.3%)	52(34.6%)	0	0	0
S3	52	89	230	141(61.3%)	87(37.8%)	0	0	2
S4	91	139	336	197(58.6%)	132(39.2%)	4(1.1%)	0	3
S5	69	119	338	219(64.7%)	118(34.9%)	0	0	1
S6	24	33	83	50(60.2%)	32(38.5%)	0	0	1
S7	26	44	93	49(52.6%)	35(37.6%)	0	0	10
S8	26	30	62	32(51.6%)	29(46.7%)	0	0	2
S9	40	22	99	77(77.7%)	21(21.2%)	0	0	1
S10	7	7	11	4(36.3%)	6(54.5%)	0	0	4
S11	12	13	20	7(35.0%)	13(65.0%)	0	0	0
S12	8	12	21	9(42.8%)	12(57.1%)	0	0	0
S13	14	12	26	14(53.8%)	12(46.1%)	0	0	0
S14	71	106	264	158(59.8%)	105(39.7%)	0	0	1
S15	182	371	1063	692(65.1%)	356(33.4%)	7(0.6%)	1(0.1%)	7
S16	24	34	98	64(65.3%)	30(30.6%)	1(1.0%)	0	3
S17	19	27	63	36(57.1%)	22(34.9%)	0	0	6
S18	67	91	232	141(60.7%)	89(38.3%)	0	0	2
S19	38	96	225	129(57.3%)	95(42.2%)	1(0.4%)	0	0
S20	64	119	214	95(44.3%)	118(55.1%)	0	0	1
S21	46	84	225	141(62.6%)	82(36.4%)	2(0.8%)	0	0
S22	15	31	54	23(42.5%)	31(57.4%)	0	0	0

Table 3.2 averages the data found in Table 3.1. The average grammar has a vocabulary of about 50 terminals, 75 nonterminals, and contains one decision requiring a lookahead depth greater than one.

Table 3.2 Average Lookahead Requirements for 22 Sample Grammars

grammar	$\|T\|$	$\|N\|$	decisions	lookahead $n \leq 3$				non-*SLL*(3)
				0	1	2	3	
Average	47.3	74.7	191.7	117(61%)	72(37.5%)	.8(.4%)	.05(.02%)	1.8(.9%)

The empirical results of Tables 3.1 and 3.2 suggest that *SLL*(k) grammars are mostly *SLL*(1). Because lookahead trees are simple, linearly sized sets of terminals when $k=1$ and because *SLL*(k) parsers are linearly sized in $|G|$, *SLL*(k) appears to be nearly a linear problem. Unfortunately, the few decisions that require $k>1$ lookahead incur the unavoidable exponential cost, $O(|T|^k)$, of computing and storing lookahead sets; more statistics are provided in Section 6.17.1.

Parser transitions generally can be made without lookahead or with a single terminal of lookahead. When more than a single terminal of lookahead is required, it is often the case that a linear approximation to the full lookahead computation can be used. This section provided empirical evidence that parsing with $k>1$ terminals of lookahead is practical because most

decisions, 98.57%, are *SLL*(0) or *SLL*(1). The complexity of the remainder of the decisions are controlled by using the linear approximation $SLL^1(k)$ as often as possible. The next section examines when *LOOK* operations are required during grammar analysis; i.e. which parser states must make lookahead decisions.

3.3.3 When Parsers Need Lookahead

LL parsers need to examine lookahead only at nonterminal decision points (on the left edge); hence, they make at most $|N|$ m-ary decisions where $|N|$ is the number of nonterminals and m is the number of productions for a particular nonterminal. *LR*-based parsers examine lookahead in any state that has an *LR*(0) *shift*/*reduce* or *reduce*/*reduce* conflict. *LL*(k) parsers' voracious appetite for lookahead arises from its need to predict which production to apply before the entire production has been scanned; *LR*(k) parsers see an entire production before deciding. In this section, we explore when parsers make lookahead decisions via an example grammar and its associated *LL*(3) and *LR*(1) machines.

Consider Grammar 3.4. The *LL*(3) machine is shown in Figure 3.11 and the *LR*(1) machine is shown in Figure 3.12 with the lookahead component missing as it is always $.

$$\begin{aligned} S &\rightarrow A\$\$ \\ A &\rightarrow abc \\ A &\rightarrow ab \\ A &\rightarrow a \end{aligned} \qquad \text{G3.4}$$

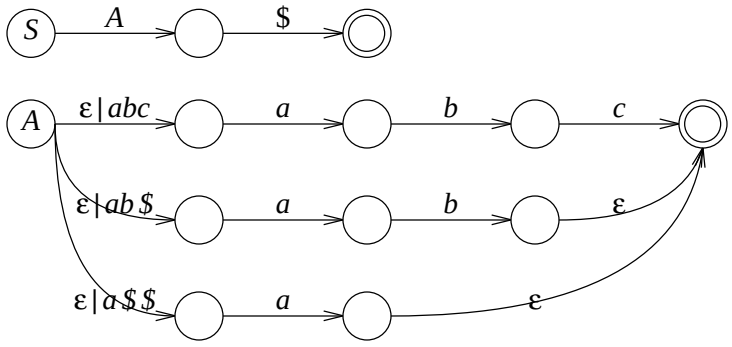


Figure 3.11 *LL*(3)-Machine for Grammar G3.4

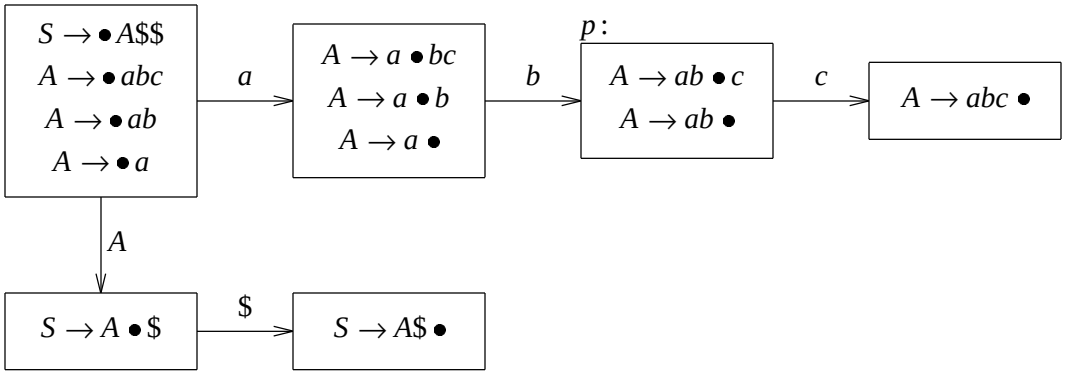


Figure 3.12 *LR* (1)-Machine for Grammar G3.4

The *LL* (3) machine makes a single decision to parse any sentence — it examines (τ_1, τ_2, τ_3) once on the left edge of *A*, making a trinary prediction. In contrast, the *LR* (1) machine, which makes decisions on the right edge, examines τ_1 once for each reduction of the productions of *A*. The *LL* (3) machine always makes exactly three token inspections to predict a production, whereas the *LR* (1) parser makes a single token inspection at each transition in the reduction of *A*.

In general, *LL* (*k*) parsers make decisions of the form:

$$A \rightarrow \bullet \alpha_1$$
$$A \rightarrow \bullet \alpha_2$$
$$\ldots$$
$$A \rightarrow \bullet \alpha_m$$

where $\alpha_i \in V^*$. *LR* (*k*) parsers examine lookahead terminals whenever an item pairs of the form:

$$A \rightarrow \alpha \bullet \beta$$
$$B \rightarrow \alpha \bullet$$

where $\alpha, \beta \in V^*$, or

$$C \rightarrow \alpha \bullet$$
$$D \rightarrow \alpha \bullet$$

are in the core of a state, where *A* and *B* are not necessarily different. Only states with at least one item of the form ‘‘$A \rightarrow \alpha \bullet$’’ are candidates for lookahead decisions. There is no such thing as a *shift*/*shift* conflict because the parser will always shift the current symbol from the input to the symbol stack until it finds a handle.

The discussion of *LL* (*k*) and *LR* (*k*) presented in this section assumes that heterogeneous automata are constructed in direct contrast to most parser generators; i.e. the left-edge *LL* (3) lookahead decision of *A* does not force all states in the parser to examine lookahead. Clearly, if only one arc emanates from a state, any prediction decision is obvious. This simple optimization

cannot be done using homogeneous automata as all states either examine lookahead or they do not.

Although $LL(k)$ and $LR(k)$ parsers have different states, examine lookahead at different times, compute lookahead from different grammar positions, and maintain different state information, lookahead states are virtually indistinguishable — both parsers' decision states map lookahead sequences to parser actions and state transitions. The next section abstracts this notion to a mathematical relation called **induces**.

3.3.4 How Parsers Use Lookahead

If one considers the mechanism by which parsers examine lookahead, as opposed to in which state and when during the parse lookahead is examined, the difference between *LL* and *LR* vanishes. Formally, any lookahead decision is a relation **induces** from $T' \subseteq T^k$ to a finite set of parser actions; e.g., ''*predict* $A \rightarrow \alpha$'' (*LL*) or ''*reduce* $A \rightarrow \alpha$'' (*LR*). In Figure 3.11, $(a,b,\$)$ **induces** ''*predict* $A \rightarrow ab$'' and, in state *p* of Figure 3.12, \$ **induces** ''*reduce* $A \rightarrow ab$''. Regardless of the range of **induces** (the set of possible parser actions), examining lookahead remains a simple relation on a subset of T^k. For example, consider the $LL(3)$ **induces** relation in Table 3.3 for state *A* of Figure 3.11.

Table 3.3 $LL(3)$ **induces** Relation for State *A* of Figure 3.11

Lookahead $(\tau_1,\tau_2,\tau_3) \in T^3$	Action
(a,b,c)	*predict* $A \rightarrow abc$
$(a,b,\$)$	*predict* $A \rightarrow ab$
$(a,\$,\$)$	*predict* $A \rightarrow a$

The lookahead 3-tuple on the left **induces** the parser actions listed on the right. Clearly, an **induces** relation, and hence a decision state, is deterministic when no lookahead **induces** more than one action — e.g., no *k*-tuple predicts more than one production (*LL*) and no *k*-tuple **induces** a *shift* and a *reduce* or **induces** a *reduce* of more than one production.

It is convenient to give each action for a given decision a unique number such that the range of **induces** is a subset of the natural numbers; **induces** then maps $T' \rightarrow \{1, 2, ..., m\}$. In the case of *LL*, *m* is the number of productions for a nonterminal. For *LR*, *m* is the number of conflicting $LR(0)$ items in a state requiring lookahead. In this manner, a generic $C(k)$ parser state can be manipulated without concern for the parser strategy.

This abstract lookahead mapping can be viewed as a *k*-dimensional vector plot. For example, consider the **induces** mapping in Table 3.4, which is plotted in Figure 3.13.

Table 3.4 Example $C(2)$ **induces** Relation

Lookahead $(\tau_1, \tau_2) \in T^2$	Action $\in \{1..3\}$
(c,b)	1
(a,e)	1
(d,c)	2
(c,d)	2
(f,c)	2
(e,a)	3

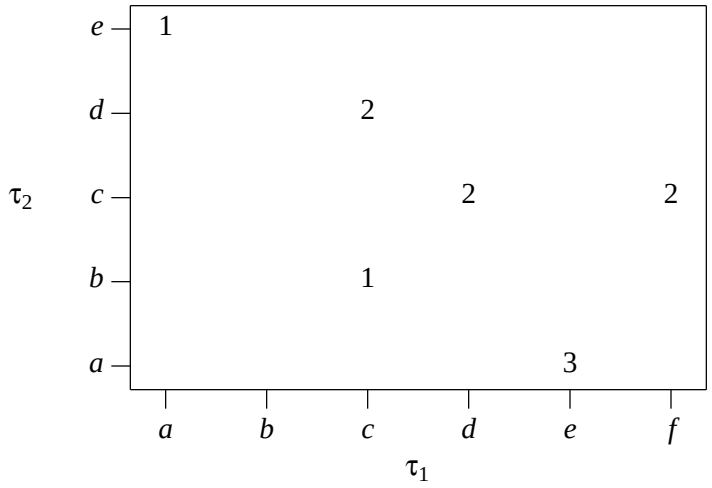


Figure 3.13 Example **induces** Relation Plot

Constructing a parser lookahead decision amounts to finding an efficient implementation of **induces** — generating a *discriminant function* that deterministically classifies a feature vector to one of *m* classes (using AI terminology). If a lookahead *k*-tuple **induces** more than one action, the underlying grammar is not deterministic; graphically, if any two vectors that map to different actions, have coincidental endpoints, no discriminant function can be found.

Observe that, when the lookahead vectors in Figure 3.13 are projected onto the τ_2 axis, a very efficient discriminant function may be found; precisely, a simple set operation on τ_2. This

operation is analogous to simplifying the relation to that of Table 3.5. According to Section 3.7.1, we may denote this type of decision as $C^1(2)$ because a lookahead depth of two is required and a 1-tuple (set) comparison is the largest atomic operation; Chapter 5 describes these decisions in more detail for $SLL^1(k)$.

Table 3.5 $LL^1(2)$ **induces** Relation

Lookahead $\tau_2 \in T$	Action $\in \{1..3\}$
$\{b,e\}$	1
$\{c,d\}$	2
$\{a\}$	3

This chapter provided a new perspective from which to view $LL(k)$ and $LR(k)$ parser construction. We developed convenient methods for representing grammars, describing parsers with heterogeneous states, and for viewing parsing decisions. Section 3.5 represented grammars as a collection of NFA, denoted GLA's, that are used to define lookahead computations in the next Chapter. Section 3.6 illustrated how parsers with states of varying complexity could be represented. In Section 3.7, we introduced linear approximate $C^1(k)$ decisions, characterized when parsers look ahead, and abstracted the notion of a parsing decision to the **induces** relation.

The $SLL(k)$ lookahead requirements for 22 sample grammars were empirically studied in Section 3.7.2. We found that 98.57% of all $SLL(k)$ decisions were correctly mapped using $k \in \{0,1\}$ and that 98.81% of all decisions that required lookahead could be mapped with a single terminal of lookahead. This empirical data supports our claim that $LL(k)$ parsers are practical because when, $k \in \{0,1\}$, $|T|^k$ is not an exponential; recall that parsers with heterogeneous states are required to take advantage of the varying lookahead requirements of decision states. One can extrapolate that $LR(k)$-based parsers are practical as well because $SLL(k)$ is weaker than $LL(k)$ and $LR(k)$ is stronger than $LL(k)$ — $LR(k)$ must surely use less lookahead than $SLL(k)$.

We abstracted parsing decisions to a mathematical relation called **induces** that summarizes a decision state mapping of lookahead sequences to parser action. The **induces** relation allows us to discuss deterministic parsers generically as they are all identical from a lookahead decision state point of view. In addition, **induces** allows us to isolate the computation of lookahead from the implementation of the mapping itself.

The following chapter examines parser lookahead, as derived from the GLA's described in this chapter, and how lookahead may be represented and computed. The cost of computing lookahead is also explored in depth.

CHAPTER 4 PARSER LOOKAHEAD

When state information alone is insufficient to determine parser action, lookahead information is used to induce the correct state change. Traditionally, lookahead information is computed, stored, and tested as sets of *k*-tuples where *k* is the lookahead depth. As discussed in this thesis, however, lookahead terminals must be considered as individual entities rather than as *k*-tuples. This necessitates a nontrivial change of perspective with regard to parser construction and grammar analysis. Pursuant to this, the previous chapter described how parsers with states of varying complexity could be described, how grammars could be stored in an advantageous manner with respect to lookahead computation, and how decision states could be abstracted to a mathematical relation called **induces** that maps lookahead symbols to parser actions; this chapter delves into the definition and representation of parser lookahead.

Although the various deterministic parsing strategies maintain different state information and use different lookahead strings, canonical lookahead operations and lookahead string representations can be very similar between strategies. The GLA grammar representation described in the previous chapter is especially convenient for computing lookahead sets. Specifically, lookahead computations may all be described as constrained walks of the collection of NFA's in the GLA; therefore, it is reasonable to view lookahead as DFA's that accept the regular language computed by the lookahead operations, which we store as child-sibling trees. We introduce $LOOK_k(p)$ as the lookahead set for grammar position or GLA state p. Similarly, we define $LOOK_k^1(p)$ as the set of terminals that can be matched k terminals in the future; we term this the ''linear approximation'' to $LOOK_k$. $LOOK_k^1$ is advantageous because it has linear time and space complexity and can be used to reduce the complexity of most lookahead decision states. We denote decisions that use $LOOK_k^1$-type information $C^1(k)$; these decisions look at most k terminals into the future and examine only 1-tuples (sets).

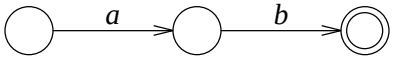
In this section, we describe efficient means for representing lookahead information (Section 4.8), define lookahead operations (Section 4.9), and provide a detailed analysis of the worst-case behavior for computing lookahead (Section 4.10).

4.1 Representation

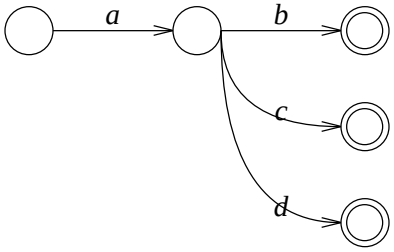
Lookahead information is normally discussed and stored as sets of *k*-tuples. Unfortunately, real programs that compute lookahead sequences cannot easily manipulate information in this form. This section introduces two alternative, equivalent structures for storing, manipulating, and examining lookahead information: deterministic finite automata (DFA's) and child-sibling trees. The DFA representation is appropriate because of its relationship to the GLA representation of a grammar. We shall view lookahead in this way, but will actually implement lookahead algorithms using child-sibling trees. Here, we show the relationship between lookahead *k*-tuples, lookahead DFA, and child-sibling trees.

Grammars are efficiently and conveniently represented by GLA because lookahead *k*-sequences for a grammar position clearly correspond to the sequence of non-ε edges along the walks of length *k* starting from the associated GLA state. Also, DFA's are more appropriate than sets of *k*-tuples for describing lookahead because of the obvious relationship between the regular lookahead languages and DFA's.

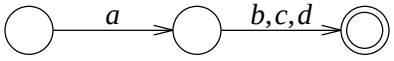
Consider the lookahead tuple (a,b); it can be trivially represented in DFA form as



A set of lookahead 2-tuples such as $\{(a,b),(a,c),(a,d)\}$ can then be represented as



where the common prefix, *a*, has been factored, thereby reducing space requirements for the lookahead information. We will sometimes use the notation

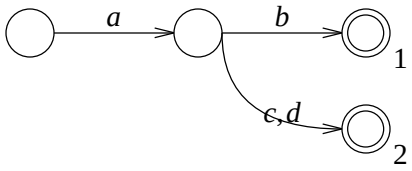


as a short form. Note that lookahead DFA's are acyclic by definition as they accept a language that is a delineation of *k*-strings; consequently, all paths are of length *k*.

When we need to discuss lookahead sets that induce different parser actions, lookahead DFA accept states will be annotated with an action number; e.g., consider Grammar G4.1, which is *LL* (2).

$$\begin{aligned} A &\rightarrow abc \\ A &\rightarrow Be \\ B &\rightarrow ac \\ B &\rightarrow ad \end{aligned} \qquad \text{G4.1}$$

The language described by this grammar is trivially $\{abc, ace, ade\}$. The 2-tuple that predicts $A \rightarrow abc$ is (a,b) and the set of 2-tuples that predicts $A \rightarrow Be$ is $\{(a,c), (a,d)\}$. In DFA form, this can be encoded as



where the subscript of *i* implies that that DFA accept-state predicts production *i*. In general, the subscript indicates which parser action to induce.

Although it is convenient to view lookahead sets as DFA's, algorithms to compute lookahead can more easily manipulate child-sibling trees. When a tree is reasonably simple, we will use the lisp notation: $(\rho\ \alpha_1\ \alpha_2\ ...\ \alpha_n)$ where ρ is the root of the tree and the α_i are the siblings, which can themselves be trees. However, this notation becomes obtuse as the size of the tree increases, therefore, trees will generally be depicted graphically. To represent *k*-tuples in graphical tree form, one performs the simple transformation in Figure 4.1.

$(a_1, a_2, ..., a_k)$	Create a subtree of the form: $(a_1\ (a_2\ (\ ...\ (a_{k-1}\ a_k))))$ a_1 $\downarrow$ a_2 $\downarrow$ $\cdots$ $\downarrow$ a_k
$\{(a_1,...), (b_1,...), ..., (c_1,...)\}$	Create a tree of the form: $(a_1\ ...)\ (b_1\ ...)\ ...\ (c_1\ ...)$ $a_1 \rightarrow b_1 \rightarrow \cdots \rightarrow c_1$ $\downarrow \quad \downarrow \quad\quad \downarrow$ $\cdots \quad \cdots \quad\quad \cdots$

Figure 4.1 Child-Sibling Tree Representation of *k*-tuple Set

Using this transformation, the 2-tuple sets above, $\{(a,b)\}$ and $\{(a,c), (a,d)\}$, can be represented as

$$\begin{array}{c} a \\ \downarrow \\ b \end{array}$$

for predicting production one and

$$\begin{array}{ccc} a & \rightarrow & a \\ \downarrow & & \downarrow \\ c & & d \end{array}$$

for predicting production two where all terminal symbols at the same lookahead depth are at the same horizontal level. As before with the DFA representation, the common left-prefix in the lookahead information can be factored out:

$$\begin{array}{ccc} a & & \\ \downarrow & & \\ c & \rightarrow & d \end{array}$$

The parallel between the DFA representation and the tree representation can be observed by rotating and flipping a tree from its normal orientation such as in Figure 4.2. Trees and acyclic DFA's are essentially duals of each other, in the graph theory sense, where the tree nodes become DFA transition labels and vice versa.

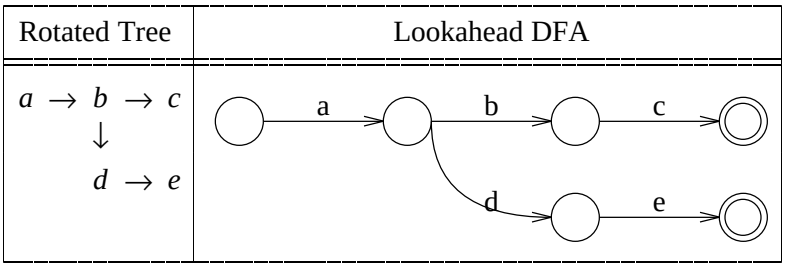


Figure 4.2 Tree and DFA Duality Example for {(a,b,c), (a,d,e)}

In the next section, we define lookahead computations and use the child-sibling tree lookahead-representation to describe these computations as GLA to set and GLA to child-sibling tree conversions.

4.2 Operations

As discussed in the previous section, we compute lookahead information by traversing GLA's. We define $LOOK_k$ to be the set of lookahead k-strings for a particular grammar position (GLA state); i.e. the set of strings that are validly recognizable, according to the parsing method, from a position by consuming exactly k terminal symbols. — set of terminal symbols that can be validly recognized, according to the parsing strategy, exactly k terminals in the ''future''. Consequently, this operation maintains a single set of terminals and has linear time and space complexity (using the correct implementation). $LOOK_k^1$ is called the ''linear approximation'' as it can be used to approximate $LOOK_k$ with a linear, covering set of k terminal sets. We define $LOOK_k$ and $LOOK_k^1$ in terms of *FIRST* and *FOLLOW* operations and in terms of GLA traversals for both $SLL(k)/SLR(k)$ and $LL(k)/LR(k)$.

4.2.1 Full Lookahead Operations

Parser lookahead information can be defined in terms of $FIRST_k$ and $FOLLOW_k$, which are insensitive to the parsing method. However, for the various parsing methods, it is convenient to have one operation that reflects the appropriate sequence of $FIRST_k$ and $FOLLOW_k$ operations necessary to compute lookahead strings for any grammar position. In this section, we define such an operation for $SLL(k)/SLR(k)$ and $LL(k)/LR(k)$ denoted $LOOK_k$. $LOOK_k$ is also defined in terms of GLA traversals for $SLL(k)/SLR(k)$.

The set of $SLL(k)/SLR(k)$ lookahead strings for a position, $A \rightarrow \alpha \bullet \beta$ is

$$LOOK_k(A \rightarrow \alpha \bullet \beta) = FIRST_k(\beta\ FOLLOW_k(A))$$

and, for $LL(k)$/$LR(k)$,

$$LOOK_k(A \rightarrow \alpha \bullet \beta) = FIRST_k(\beta\ \gamma) \text{ where } S \Rightarrow^*_{lm,rm} wA\gamma$$

with $w \in T^*, \gamma \in V^*$ for $LL(k)$ and $w \in V^*, \gamma \in T^*$ for $LR(k)$; $LR(k)$ uses $\Rightarrow^*_{rm}$ and $LL(k)$ uses $\Rightarrow^*_{lm}$.

The definition of $LOOK_k(p)$, for some grammar position p, can be described in terms of specific walks beginning at the node in the GLA created for position p. Figure 4.3 provides a set of recurrences that describe $LOOK_k$ trees.

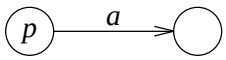
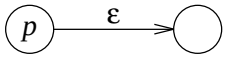
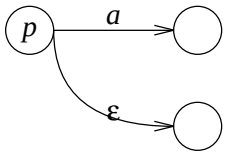
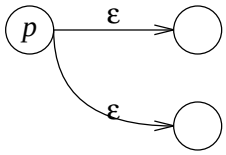
GLA Fragment	$LOOK_k(p)$ Operation
	$\begin{cases} a & k=1 \\ a & \\ \downarrow & k>1 \\ LOOK_{k-1}(p \rightarrow edge_1) & \end{cases}$
	$LOOK_k(p \rightarrow edge_1)$
	$\begin{cases} a \rightarrow LOOK_k(p \rightarrow edge_2) & \\ \downarrow & k=1 \\ LOOK_{k-1}(p \rightarrow edge_1) & \\ LOOK_{k-1}(p \rightarrow edge_1) \rightarrow LOOK_k(p \rightarrow edge_2) & k>1 \end{cases}$
	$LOOK_k(p \rightarrow edge_1) \rightarrow LOOK_k(p \rightarrow edge_2)$

Figure 4.3 *SLL* (*k*) $LOOK_k$ Operations on GLA

where only two edges, $p \rightarrow edge_1$ and $p \rightarrow edge_2$, emanate from a GLA node; $p \rightarrow edge_1$ is the edge pointing from left to right and $p \rightarrow edge_2$ is the edge pointing downward.

Algorithms for $LOOK_k$ are provided in Section 6.16; $LOOK_k$ is used by parser construction algorithms in Chapter 6. Unfortunately, the use of $LOOK_k$ is expensive due to the exponential nature of lookahead information. In an effort to reduce the need for $LOOK_k$ operations, we define a linear, covering approximation called $LOOK_k^1$ that can often be used by in its place.

4.2.2 Linear, Approximate, Lookahead Operations

An attempt is made to resolve parser nondeterminisms with as simple a lookahead decision as possible. Most decisions can be made with no lookahead or with a single terminal of lookahead. Of the decisions that require more than one lookahead terminal, it is often the case that terminals at certain depths, rather than terminal sequences, can be used to distinguish between parser transitions. Storing the lookahead k-tuples for a parser decision state has exponential complexity whereas storing the terminals that can appear at the various depths requires only k sets of maximum size $|T|$. We define parsers that make only terminal set (1-tuple) comparisons and look at most k terminals into the future as $C^1(k)$. The corresponding lookahead computation is denoted $LOOK_k^1$.

Denote the set of $SLL^1(k)/SLR^1(k)$ lookahead strings for a position, $A \rightarrow \alpha \bullet \beta$ as

$$LOOK_k^1(A \rightarrow \alpha \bullet \beta) = FIRST_k^1(\beta\ FOLLOW_k(A))$$

and, for $LL^1(k)/LR^1(k)$,

$$LOOK_k^1(A \rightarrow \alpha \bullet \beta) = FIRST_k^1(\beta\ \gamma) \text{ where } S \Rightarrow_{lm,rm}^* wA\gamma$$

Again, $LR(k)$ would use $\Rightarrow_{rm}^*$ and $LL(k)$ would use $\Rightarrow_{lm}^*$.

The normal lookahead operations are modified in the following way:

$$FIRST_k^1(\alpha) = \{\ a \mid \alpha \Rightarrow^* w \text{ and } w = xay \text{ for } x \in T^{k-1}\ \}$$

and

$$FOLLOW_k^1(A) = \{\ FIRST_k^1(\beta) \mid S \Rightarrow^* \alpha A \beta\ \}$$

where $a \in T$, $y \in V^*$, and $\alpha, \beta \in V^*$. $FIRST_k^1$ is the end-of-file marker, \$, when a w of the form xay cannot be found.

As with $LOOK_k$, $LOOK_k^1$ is easily defined as a collection of GLA traversals. Figure 4.4 gives a set of recurrences that describe $LOOK_k^1$ sets.

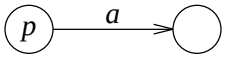
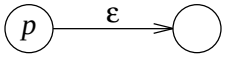
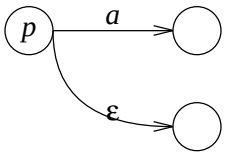
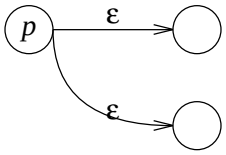
GLA Fragment	$LOOK_k^1(p)$ Operation
	$\begin{cases} a & k=1 \\ LOOK_{k-1}^1(p \rightarrow edge_1) & k>1 \end{cases}$
	$LOOK_k^1(p \rightarrow edge_1)$
	$\begin{cases} a \bigcup LOOK_k^1(p \rightarrow edge_2) & k=1 \\ LOOK_{k-1}^1(p \rightarrow edge_1) \bigcup LOOK_k^1(p \rightarrow edge_2) & k>1 \end{cases}$
	$LOOK_k^1(p \rightarrow edge_1) \bigcup LOOK_k^1(p \rightarrow edge_2)$

Figure 4.4 $SLL^1(k)$ $LOOK_k^1$ Operations on GLA

where only two edges, $p \rightarrow edge_1$ and $p \rightarrow edge_2$, emanate from a GLA node; $p \rightarrow edge_1$ is the edge pointing from left to right and $p \rightarrow edge_2$ is the edge pointing downward.

The reader may argue that it is more efficient to have $LOOK_k^1$ compute sets at all depths up to k rather than just for depth k. However, because we expect most decisions to require a single token of lookahead, computing all sets up to depth k unnecessarily complicates the definition, algorithm and implementation.

$LOOK_k^1$ is a linear approximation to $LOOK_k$ that is often sufficient to induce correct parser action. Its two advantages are that it reduces grammar analysis to a potentially linear complexity and parser decision states can be stored in space $O(|T| \times k)$ rather than $O(|T|^k)$. Parsing with these compressed lookahead sets is explored in detail in Chapter 5.

The recurrences in Figures 4.4 and 4.3 are simple, but do not take into account the fact that a lookahead operation may arrive back at a previously-visited node. Cycles are not too much of a problem unless the results of lookahead computations need to be saved. Storing incomplete information can be difficult and, therefore, in the next section, we study lookahead computation cycles, which cause early termination of computations.

4.2.3 Lookahead Computation Cycles

A context-free language cannot be represented exactly with a GLA, but any finite set of substrings of the language generated by a CFG is regular and, hence, can be described by a DFA. The bounded lookahead information for any position in the grammar is such a language; therefore, it is reasonable to represent a grammar as a large, intertwined, collection of NFA's. Computing lookahead information is then a simple matter of performing a constrained traversal of the GLA; the computations are similar to NFA *REACH* and ε-*CLOSURE* operations. From a graph theory viewpoint, one is recording all walks of length k beginning at a particular GLA state, where ε-edges count as length zero. When the lookahead languages of a grammar are represented by a GLA, $FIRST_k$ and $FOLLOW_k$ operations become the same computation except that $FIRST_k$ begins at nonterminal entry positions and $FOLLOW_k$ begins at nonterminal exit positions (recall that nonterminal exit states point to all states that following references that nonterminal).

Lookahead information is straightforward to compute for many grammars because there are no GLA cycles. However, If a $LOOK_k$ or $LOOK_k^1$ computation were to reach a state that is currently a member of a walk in progress (for the same k), a cycle would have occurred. We define a cycle as any lookahead computation recurrence of the form

$$LOOK_n \leftarrow f(LOOK_n)$$

for some $n \leq k$ and some computation f. An algorithm must not pursue this type of redundant computation in order to terminate. This section describes the difficulties and semantics behind cycles in *LOOK* computations. We begin by describing what computation cycles mean for the various parsing strategies and then present $FIRST_k$ and $FOLLOW_k$ cycle examples.

Cycle detection is important from a computation caching standpoint because early termination of a lookahead computation due to a cycle yields incomplete information that must not be cached as complete. Therefore, since our *LOOK* algorithms will cache results computed for both nonterminal entry and exit states, we must consider cycles discovered from both state types. Note that inefficient algorithms that do not save the results of previous computations only need to worry about cycles causing nontermination of the algorithm.

In the $SLL(k)/LL(k)$ $FIRST_k$ sense, cycles are direct or indirect left-recursion and render the grammar non-$SLL(k)/LL(k)$. With regards to $LR(k)$, cycles cannot occur as lookahead operations are confined to string append and $FIRST_k$ operations; $SLR(k)$ analysis runs the same risk as $LL(k)$ in terms of nontermination, but *LR*-based parsers do not consider left recursion an error.

4.2.3.1 Example $FIRST_k$ Cycle

Cycles in the $FIRST_k$ sense arise from recursive grammar productions such as those in Grammar G4.2:

$$\begin{aligned} A &\rightarrow aB \\ A &\rightarrow B \\ B &\rightarrow A \\ B &\rightarrow b \end{aligned} \qquad \text{G4.2}$$

The GLA for Grammar G4.2 is shown in Figure 4.5.

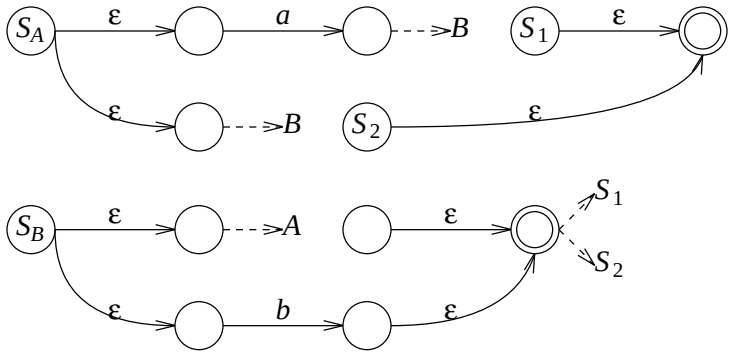


Figure 4.5 Grammar With $FIRST_k$ Cycle

$LOOK_2(A \rightarrow \bullet aB)$ discovers a cycle:

$$LOOK_2(A \rightarrow \bullet aB) = \overset{a}{\downarrow} \; LOOK_1(B \rightarrow \bullet A) \rightarrow LOOK_1(B \rightarrow \bullet b)$$

$$LOOK_1(B \rightarrow \bullet A) = LOOK_1(A \rightarrow \bullet aB) \rightarrow LOOK_1(A \rightarrow \bullet B)$$
$$LOOK_1(A \rightarrow \bullet B) = LOOK_1(B \rightarrow \bullet A) \rightarrow LOOK_1(B \rightarrow \bullet b)$$

$LOOK_1(B \rightarrow \bullet A)$ requires itself to complete the computation. This cycle indicates that Grammar G4.2 is left recursive. Notice that $LOOK_2(B \rightarrow \bullet aB)$ requires $LOOK_1(B \rightarrow \bullet aB)$, but because $LOOK_2$ and $LOOK_1$ are different computations entirely, this does not constitute a cycle. A computation at depth n can never attempt a computation at depth $n+1$ because $LOOK_k$ is a monotonically decreasing function of k.

One may view cycles more clearly in a computation dependence graph such as the one depicted for Grammar G4.2 in Figure 4.6.

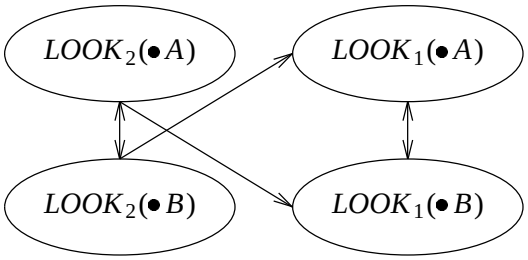


Figure 4.6 Partial Computation Dependence Graph for Grammar G4.2

where an edge from computation $LOOK_1(\bullet A)$ to $LOOK_1(\bullet B)$ indicates that $LOOK_1(\bullet A)$ depends on $LOOK_1(\bullet B)$ to complete its computation and $LOOK_k(\bullet A)$ indicates the combined $LOOK_k$ for all productions of nonterminal *A*. Therefore, it is always the case that dependence arcs move vertically, move from left to right, or loop on a state; dependence arcs never point from right to left.

4.2.3.2 Example $FOLLOW_k$ Cycle

Cycles found during any $FOLLOW_k$-type *LOOK* operation are not a problem from a lookahead definition point of view because a cycle in this case means simply that that computation result has already been included in the set of possible *k*-strings.

LOOK computations may continue past the GLA accept state of a nonterminal, thus, beginning a *FOLLOW* operation. If a $LOOK_n$ operation on some GLA state eventually returns to that same state and requires a *LOOK* operation for the same *n*, $LOOK_n$ has detected a cycle; that branch of the computation must terminate. Grammar G4.3 contains a cycle in the *FOLLOW* sense.

$$\begin{aligned} A &\rightarrow a_1 B a \\ A &\rightarrow a_2 B \\ B &\rightarrow a_3 C \\ C &\rightarrow a_4 A \\ C &\rightarrow \end{aligned} \qquad \text{G4.3}$$

Computing $LOOK_1(C \rightarrow a_3 C \bullet)$ requires $LOOK_1(C \rightarrow a_3 C)$ to complete. $LOOK_1(C \rightarrow a_3 C)$ is a member of a computation cycle and any member of this cycle has the same $LOOK_1$ set; thus, $LOOK_1$ for any accept state is $\{a\}$. This can be easily shown by the transitive property of assignment:

$$\begin{aligned} LOOK_1(B \rightarrow a_3 C \bullet) &= LOOK_1(A \rightarrow a_2 B \bullet) \rightarrow a \\ LOOK_1(A \rightarrow a_2 B \bullet) &= LOOK_1(C \rightarrow a_4 A \bullet) \\ LOOK_1(C \rightarrow a_4 A \bullet) &= LOOK_1(B \rightarrow a_3 C \bullet) \end{aligned}$$

By expanding any computation in the cycle, any $LOOK_1$ computation reduces to

$$LOOK_1(p) = LOOK_1(p) \rightarrow a$$

for some *p*, which is simply

$$LOOK_1(p) = a$$

The computation dependencies are partially depicted in Figure 4.7.

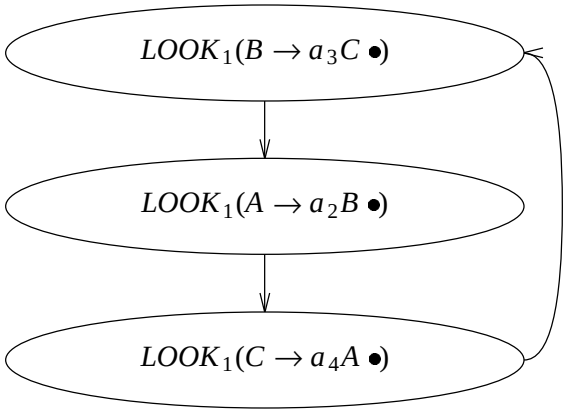


Figure 4.7 Partial Computation Dependence Graph for Grammar G4.3

Lookahead computations typically request many other computations that, in turn, invoke others. This is one source of lookahead computation exponentiality with the size of $LOOK_k$ lookahead information being the other. The next section explores the worst-case behavior of lookahead computations as we have defined them in this chapter.

4.3 Complexity of Lookahead Information Computation

The previous methods for $C(k)$ grammar analysis do not explicitly compute lookahead sets; instead, they test small pieces of the associated canonical parsers with each permutation of T^k — a clearly impractical method. Our techniques have the advantage that they compute lookahead sets directly, which are needed for parser construction, and they have better average performance, although they are little better in worst-case complexity. This section describes the worst-case complexity in time and space of our lookahead computation algorithms; the discussion valid for algorithms in Chapters 5, 6, and 7.

There are a fixed number of possible lookahead computations possible for a given grammar, $O(|G| \times k)$, because there are k $LOOK_k$ and $LOOK_k^1$ operations defined for all $|G|$ positions in the grammar. However, without results caching, these computations can be computed multiple times, which renders grammar analysis exponential in nature, $O(|G|^k)$, for one computation on a node. To illustrate this, we present a three-dimensional computation space where each of k planes has a copy of the grammar in GLA form with the lowest plane associated with k=1; see Figure 4.8. Computation may proceed within one plane and may dip down into lower planes, but may never jump up to another level ($LOOK_k$ cannot invoke $LOOK_{k+1}$).

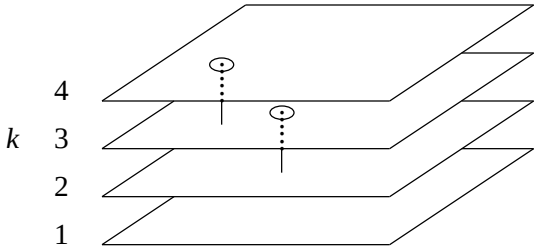


Figure 4.8 $LOOK_k$ Computation Planes

One computation at the k=1 plane has at most $|G|$ nodes to visit and, hence, has time complexity $O(|G|)$. Each of the $|G|$ nodes in plane k=2 could make a computation in plane k=1, yielding a time complexity for the k=2 plane of $O(|G|^2)$ for one computation. In general at plane k, there are $O(|G|^{k-1})$ operations possible at all lower levels and $|G|$ nodes can be visited by each computation on plane k; One computation at level k is then $O(|G|^k)$. A grammar which exhibits this exponentiality is Grammar G4.7.

$$\begin{aligned} A &\rightarrow aA \\ A &\rightarrow bA \\ A &\rightarrow cA \\ A &\rightarrow d \end{aligned} \qquad \text{G4.7}$$

The left edge of each production of nonterminal A is visited for each reference to A and for each k; each invocation of A can ‘‘fork’’ 3 other invocations. Empirical results suggest that the number of uncached $LOOK_k^1$ computations on A is approximately 3^k where the number 3 arises from the three references to A. For example, $LOOK_1^1(A)$ reports that exactly one $LOOK_1^1(A)$ was invoked because none of the A references were seen. $LOOK_2^1(A)$ reports that 4 computations where requested: one for the initial invocation and one for each reference to A in A. Figure 4.9 demonstrates that the number of $LOOK_k^1$ computation requests is, indeed, exponential in the size of the grammar.

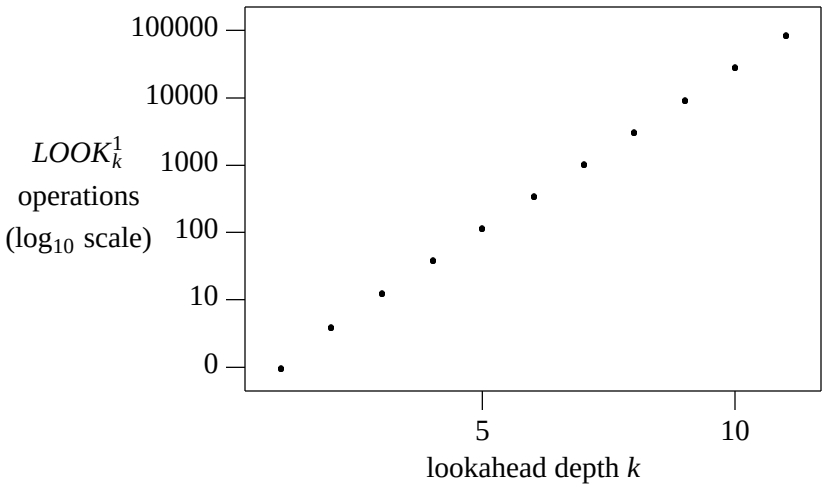


Figure 4.9 Number of $LOOK_k^1$ Invocations for Grammar G4.7 (no caching)

When the result of every *LOOK* computation is cached, the number of computations performed is limited to the number of computations defined on the grammar — $O(|G| \times k)$. The time to fill all caches with information is proportional, then, to the number of defined computations. Let each operation within a node, such as a cache store, takes time proportional to $O(C_{info})$, which is the time required to do an operation on the lookahead information. The cache removes redundant computations; the number of cache lookups is proportional to the number of nonterminal references times the lookahead depth, or $O(|G| \times k)$, in the worst-case.

The space required to compute all *LOOK* computations is proportional to the maximum runtime stack-depth of the algorithm and the space required for the cache. The maximum number of recursive invocations of a *LOOK* computation is limited to $|G| \times k$ because a node/arc may be visited/traversed at most once per lookahead depth. The cache, on the other hand, can become large as it is proportional to $|G| \times k \times C_{info}$.

Combining this information, we observe that the time to fill the cache and to access it the maximum number of times is proportional to $O(|G| \times k \times C_{info} + |G| \times k)$ and that the space required to perform this is $O(|G| \times k + |G| \times k \times C_{info})$. For the $LOOK_k^1$ computations, which compute the single set of terminals visible at depth k, C_{info} is proportional to the size of a set, $|T|$. For $LOOK_k$, the lookahead is a tree composed to terminals of depth k, which yields a C_{info} of $|T|^k$.

Summarizing, the worst-case complexity to compute all possible $LOOK_k^1$ operations is $O(|G| \times k \times |T|)$ in both time and space. To compute all $LOOK_k$ trees, the worst-case time and space complexity is $O(|G| \times k \times |T|^k)$.

This chapter described lookahead computations, lookahead information representation, and lookahead computation complexity. Our approach to grammar analysis consists of simple

operations on GLA for computing lookahead. In Section 4.9.1, we defined $LOOK_k(p)$, the set of lookahead strings recognizable from GLA state or grammar position p, for the various deterministic parsing strategies. A similar operation, $LOOK_k^1$, was defined in Section 4.9.2 that approximates the exponential $LOOK_k$ information with compressed lookahead information of linear size; $LOOK_k^1$ has the advantage that it is linear to compute and yields a much smaller amount of information. Specifically, $LOOK_k^1$ computes the set of all terminals that can be recognized exactly k terminals in the future. Section 4.8 described how k-tuples are best viewed as DFA's due to our representation of grammars as GLA. It further illustrated that child-sibling trees, which are duals of lookahead DFA's, are very convenient in practice for manipulating lookahead information.

The worst-case computation of lookahead information was presented as an exponential function of k due to the size of lookahead information for a fixed grammar. The occurrence of this worst-case behavior can be reduced, however, through the use of $C^1(k)$ approximations to the full lookahead information. $LOOK_k^1$ was introduced as the set of terminals that can be recognized k terminals in the future; $LOOK_k^1$ has linear time and space complexity.

The next two chapters (5 and 6) use the observations and information representations presented in this chapter to construct $SLL^1(k)$ and $SLL(k)$ parsers in their entirety where $SLL^1(k)$ is an $SLL(k)$ parser which uses only $C^1(k)$ decision templates. They provide algorithms to compute lookahead in the $SLL^1(k)$ and $SLL(k)$ sense and to test for the $SLL^1(k)$ and $SLL(k)$ property; construction of these parsers is also described. Chapter 7 examines the rest of the $LL(k)$ and $LR(k)$ hierarchy and generalizes $C^1(k)$ to $C^m(k)$ where m is the size of the largest tuple comparison.

CHAPTER 5 $SLL^1(k)$ — A LINEAR APPROXIMATION TO $SLL(k)$

The set of $SLL(k)$ lookahead sequences of length k for any grammar position form a finite, regular language with $O(|T|^k)$ sentences in the worst case. By employing the minimum necessary lookahead depth, k, for each decision in an $SLL(k)$ parser, this exponentiality can be reduced or avoided in many cases. When a decision does require $k>1$, it is often sufficient to examine the set of symbols visible at certain lookahead depths rather than complete k-sequences. In this chapter, we define the $SLL^1(k)$ parser class whose decisions look at most k terminals into the future and consider set membership tests (1-tuple comparisons) to be the largest atomic operation. $SLL^1(k)$ is an approximation to $SLL(k)$ that has linear grammar analysis and lookahead information size characteristics. Moreover, empirical results indicate that $SLL^1(k)$ covers about 75% of all $SLL(k)$ decisions for $k>1$ and 99% of all $SLL(k)$ decisions for $k\geq 1$; see Section 5.11.2.

This chapter describes $SLL^1(k)$ in its entirety, from grammar analysis to parser construction. Section 5.1 introduces $SLL^1(k)$ by way of an example, provides empirical studies that show $SLL^1(k)$ to cover a significant fraction of the $SLL(k)$ decisions, and then formalizes the approach. Section 5.2 examines $SLL^1(k)$ lookahead information and computations in detail. Using this lookahead analysis, Section 5.3 presents an algorithm to test for the $SLL^1(k)$ property. The implementation of the associated **induces** relation (parsing decision state) is described in Section 5.4.

5.1 $SLL^1(k)$ Decisions

$SLL^1(k)$ is strictly weaker than $SLL(k)$ for $k>1$ because it considers k terminal symbols individually rather than k-sequences; lookahead space requirements are, therefore, $O(|T|\times k)$ rather than $O(|T|^k)$. $SLL^1(k)$ lookahead information is comprised of the sets of all terminals visible at each of k lookahead depths. Specifically, let Λ (an array of sets) represent the linear-approximate lookahead information for a particular production. Λ_i is the collection of all terminals visible at depth i starting from that grammar position. If the Λ sets for two productions do not have at least one disjoint lookahead depth, the two productions are not separable — the decision is not $SLL^1(k)$ deterministic.

The high compression of Λ sets results in a reduction in recognition strength because most sequence information is no longer available. With regards to $SLL^1(k)$ decisions, the discriminating factor between **induces** actions is not specific terminal sequences, but sets of terminals at

certain lookahead depths. As one might expect for a lookahead depth of one, $SLL^1(1)$ and $SLL(1)$ are equivalent. The following four subsections provide an example $SLL^1(k)$ grammar, give $SLL^1(k)$ decision statistics, discuss lookahead information compression, and formalize $SLL^1(k)$ determinism.

5.1.1 Example $SLL^1(k)$ Grammar

In order to illustrate the difference between $SLL^1(k)$ and $SLL(k)$, we present an example, which is $SLL(2)$, but is also $SLL^1(2)$. It provides some insight as to why $SLL^1(2)$ is effective and simpler than $SLL(k)$. Consider the recognition of programming language labels presented in the Grammar G5.1.

$$\begin{aligned} S &\rightarrow wS \\ S &\rightarrow waE \\ S &\rightarrow ilErS \\ S &\rightarrow fwaEtE \end{aligned} \qquad \text{G5.1}$$

where *waE* specifies that a word followed by an assignment operator followed by an expression should be matched; *i*, *f*, *t*, *a*, *l*, and *r* are ‘‘if’’, ‘‘for’’, ‘‘to’’, ‘‘assign’’, ‘‘(’’, and ‘‘)’’ ‘‘respectively. The first two productions of the statement nonterminal symbol, *S*, can begin with a word, *w*. Hence, this grammar is not $SLL(1)$. Lookahead of depth two is required, but 2-tuple comparisons are unnecessary. To demonstrate this, we provide the $SLL(2)$ solution and then contrast it with the $SLL^1(2)$ solution.

Consider the $SLL(2)$ lookahead vectors (τ_1, τ_2) plotted for grammar G5.1 in Figure 5.1.

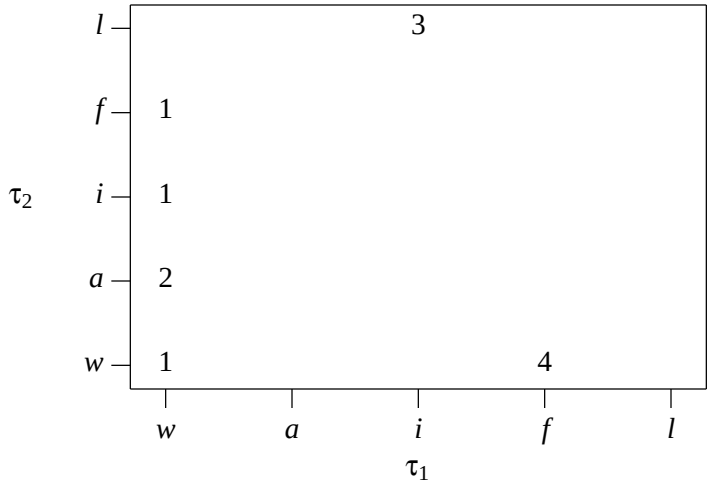


Figure 5.1 Lookahead Vector Plot for Grammar G5.1

The vector endpoints in the lookahead vector plot specify the production number that the vector predicts. The associated *SLL* (2) decision can be represented by the **induces** relation shown in Table 5.1.

Table 5.1 *SLL* (2) **induces** Relation for Grammar G5.1

Lookahead $(\tau_1, \tau_2) \in T^2$	Action
(w,w)	*predict* $S \rightarrow wS$
(w,i)	*predict* $S \rightarrow wS$
(w,f)	*predict* $S \rightarrow wS$
(w,a)	*predict* $S \rightarrow waE$
(i,l)	*predict* $S \rightarrow ilErS$
(f,w)	*predict* $S \rightarrow fwaEtE$

Equivalently, the decision can be represented in heterogeneous state form such as that in Figure 5.2.

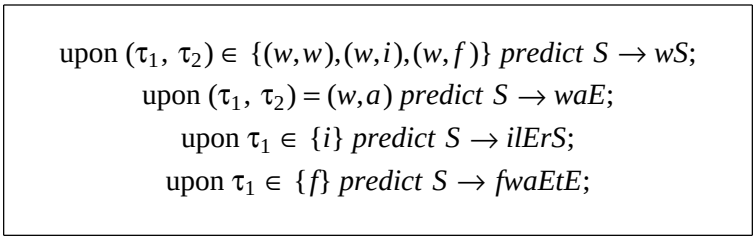


Figure 5.2 Automaton for **induces** in Table 5.1

By inspection of Figure 5.1, one finds that a series of two set membership tests can be used to map lookahead vectors to an action in the image. Lookahead depth τ_1 uniquely maps (i,l) and (f,w) to actions 3 and 4, but cannot separate actions 1 and 2. However, once actions 3 and 4 have been removed from consideration, a second set membership test on τ_2 can be used to separate actions 1 and 2. With this in mind, the *SLL* (2) **induces** relation can be represented in a functionally equivalent, but compressed, manner by the **induces** relation of Table 5.2.

Table 5.2 $SLL^1(2)$ **induces** Relation for Grammar G5.1

Lookahead $\tau_1, \tau_2 \in T,T$	Action
$\{w\}, \{w,i,f\}$	*predict* $S \rightarrow wS$
$\{w\}, \{a\}$	*predict* $S \rightarrow waE$
$\{i\}, \{l\}$	*predict* $S \rightarrow ilErS$
$\{f\}, \{w\}$	*predict* $S \rightarrow fwaEtE$

The **induces** relation in Table 5.2 is advantageous for two reasons: These k sets are easier to compute than $|T|^k$ tuples and $SLL^1(k)$ decisions can be implemented more practically. For example, the new relation can be represented in heterogeneous state form as in Figure 5.3.

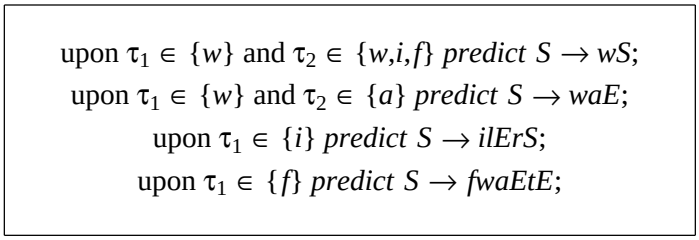


Figure 5.3 Automaton for **induces** in Table 5.1

The $SLL(2)$ and $SLL^1(2)$ automaton states appear to be equally complex, but, in general, $SLL(k)$ **induces** relations (and resulting parser decision states) will be exponential in size whereas $SLL^1(k)$ lookahead information is only $O(|T| \times k)$.

5.1.2 Empirical Studies of $SLL^1(k)$ Versus $SLL(k)$

To examine the recognition strength of $SLL^1(k)$ relative to $SLL(k)$, we examined 22 sample grammar supplied by PCCTS [PDC92] users; see the Appendix for a description of the grammars. Although ANTLR (the parser generator in PCCTS) generates $LALL(k)$ parsers and allows semantic predicates (semantics may alter the parse), the grammars still provide useful information regarding the relationship between compressed and full lookahead.

Table 5.3 provides data collected for 22 sample grammars and breaks down the lookahead decisions by type (either $SLL^1(k)$ or $SLL(k)$). There are no $SLL(1)$ decisions because $SLL^1(1)$ and $SLL(1)$ are identical — they both compute the set of terminals that can be matched next. Most decisions can be handled by $SLL^1(k)$ for $k>1$. Also, note that most decisions need only zero or one terminal of lookahead.

Table 5.3 Deterministic Lookahead Requirements By Decision Type for 22 Sample Grammars

grammar	decisions	lookahead $k \leq 3$			
		SLL (0)	$SLL^1(k)$: 1 *SLL* (*k*): 1	2 2	3 3
S1	311	198	107 0	2 1	0 0
S2	150	98	52 0	0 0	0 0
S3	230	141	87 0	0 0	0 0
S4	336	197	132 0	3 1	0 0
S5	338	219	118 0	0 0	0 0
S6	83	50	32 0	0 0	0 0
S7	93	49	35 0	0 0	0 0
S8	62	32	29 0	0 0	0 0
S9	99	77	21 0	0 0	0 0
S10	11	4	6 0	0 0	0 0
S11	20	7	13 0	0 0	0 0
S12	21	9	12 0	0 0	0 0
S13	26	14	12 0	0 0	0 0
S14	264	158	105 0	0 0	0 0
S15	1063	692	356 0	4 3	1 0
S16	98	64	30 0	1 0	0 0
S17	63	36	22 0	0 0	0 0
S18	232	141	89 0	0 0	0 0
S19	225	129	95 0	1 0	0 0
S20	214	95	118 0	0 0	0 0
S21	225	141	82 0	2 0	0 0
S22	54	23	31 0	0 0	0 0

Table 5.4 summarizes the number of nondeterministic decisions for lookahead depths greater than one. The number of non-$SLL(3)$ decisions arise from three areas. First, the grammars were taken from users of ANTLR, an $LALL(k)$ parser generator ($SLL(k) \subset LALL(k)$ [SiS82]). Second, some of the grammars used ANTLR's semantic predicates to resolve syntactic nondeterminisms with semantic information, which is unavailable to our $SLL(k)$ grammar analysis tool. Third, a number of nondeterminisms arise normally in real grammars; e.g., the infamous ''dangling-else'' construct.

Because $SLL^1(k)$ has linear time and space complexity, it is efficient to make k very large to see if the extra lookahead will resolve any $SLL(k')$ decisions for $k'<k$. The ''$SLL(3)$, non-$SLL^1(10)$'' column illustrates that only a handful of decisions must use full $SLL(k)$ lookahead information. The ''$SLL^1(10)$, non-$SLL(3)$'' column indicates that one of the sample grammars had three decisions that linear $SLL^1(10)$ could resolve, but $SLL(3)$ could not. Clearly, $SLL(10)$ could resolve the nondeterminism, but computing $O(|T|^{10})$ 10-tuples would take lifetimes to terminate. Hence, in practice, there is no strict ordering between $SLL(k)$ and its linear approximation $SLL^1(k)$ because there is a limit to the depth of lookahead available to $SLL(k)$ decisions.

Table 5.4 Number Nondeterministic Lookahead Decisions By Type for 22 Sample Grammars

grammar	number of decisions				
	SLL (2), *SLL* (3)	non-*SLL* (3)	non-*SLL* [1](10)	*SLL* (3), non-*SLL* [1](10)	*SLL* [1](10), non-*SLL* (3)
S1	3	3	4	1	0
S2	0	0	0	0	0
S3	0	2	2	0	0
S4	4	3	4	1	0
S5	0	1	1	0	0
S6	0	1	1	0	0
S7	0	10	10	0	0
S8	0	2	2	0	0
S9	0	1	1	0	0
S10	0	4	4	0	0
S11	0	0	0	0	0
S12	0	0	0	0	0
S13	0	0	0	0	0
S14	0	1	1	0	0
S15	8	7	10	3	0
S16	1	3	0	0	**3**
S17	0	6	6	0	0
S18	0	2	2	0	0
S19	1	0	0	0	0
S20	0	1	1	0	0
S21	2	0	0	0	0
S22	0	0	0	0	0

To summarize the information concerning decision types, Table 5.5 sums the columns in Table 5.4.

Table 5.5 Total Number Nondeterministic Lookahead Decisions By Type for 22 Sample Grammars

grammar	number of decisions					
	SLL(1)	*SLL*(2), *SLL*(3)	non-*SLL*(3)	non-SLL^1(10)	*SLL*(3), non-SLL^1(10)	SLL^1(10), non-*SLL*(3)
Total	1584	19	47	49	5	3

There were 19 decisions requiring *SLL*(2) and *SLL*(3) of which 5 could not be handled by $SLL^1(k)$ for k up to 10; hence, 5/19×100 = 26% decisions required full *SLL*(k) lookahead information or 74% of all sample decisions, which require $k>1$ lookahead, can be handled by SLL^1(10). Again, there are even some decisions that $SLL^1(k)$ can handle that *SLL*(k) cannot due to exponential computation complexity restrictions on *SLL*(k). The next section explores in more the detail the reduction in strength from *SLL*(k) to $SLL^1(k)$ for the same k.

5.1.3 Recognition Strength Versus Space Requirements

$SLL^1(k)$ is an proper subset of *SLL*(k) because there are *SLL*(k) **induces** that cannot be mapped correctly with $SLL^1(k)$. The merging of all terminals at lookahead depth i to create the Λ_i sets generates *artificial lookahead vectors* because most terminal sequence information has been destroyed. $O(|T|^k)$ sequences of terminals have been compressed to k sets of terminals, which comes at the cost of reduced recognition strength. A series of k set membership tests effectively matches any permutation of terminals formed by the concatenation of a terminal from Λ_1 followed by a terminal from Λ_2 and so on. For example, the two lookahead tuples (a,b) and (c,d) for some action j have sets $\Lambda_1^j = \{a,c\}$ and $\Lambda_2^j = \{b,d\}$. The $SLL^1(2)$ decision strategy, however, maps any tuple with $\{a,c\}$ at lookahead depth one and $\{b,d\}$ at lookahead depth two to action j — tuples (a,b), (a,d), (c,b), and (c,d). Another way to view this compression is to consider the representative DFA's such as those depicted in Figure 5.4.

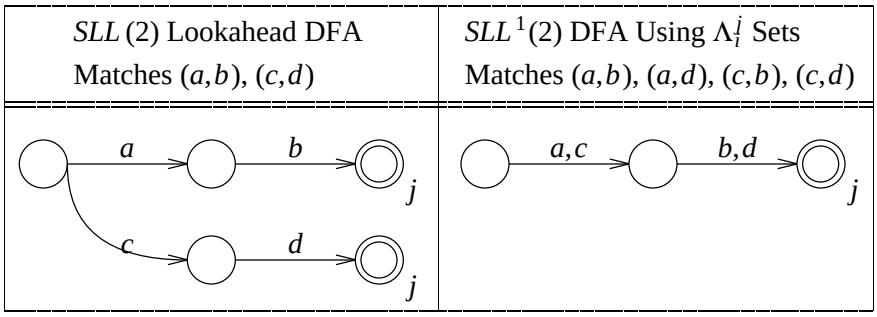


Figure 5.4 DFA's for Example Lookahead Tuple Space

In terms of lookahead vector plots, Λ set compression projects all *k*-vectors onto the τ_i axes. For example, consider the **induces** relation of Table 5.6, which results in the lookahead vector plot shown in Figure 5.5.

Table 5.6 Example *SLL* (2) **induces** Relation

Lookahead $(\tau_1,\tau_2) \in T^2$	Action $\in \{1..4\}$
(b,b)	1
(c,e)	1
(d,d)	1
(c,a)	2
(e,c)	2
(a,b)	3
(a,d)	3
(d,c)	4

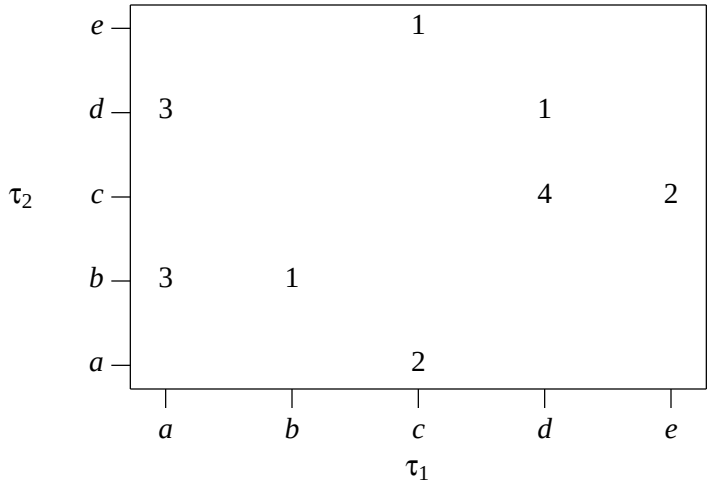


Figure 5.5 Example Lookahead Vector Plot

The associated **induces** relation is *SLL* (2) because no 2-tuple induces more than a single action (there are no overlapping vector endpoints in the lookahead plot). The relation is $SLL^1(2)$ because there exists a τ_i axis upon which, for each pair of actions, all lookahead vectors for those pairs may be projected without overlap. Action 3 is projected without overlap onto the τ_1 axis, effectively separating it from the other three actions; similarly, actions 1 and 2 are separable when projected onto the τ_2 axis. An $SLL^1(2)$ **induces** is therefore consistent and is shown in Table 5.7. Note that there need not be a single lookahead depth that separates all productions; it is sufficient to have a depth that separates each pair of productions.

Table 5.7 Example $SLL^1(2)$ **induces** Relation

Lookahead $\tau_1, \tau_2 \in T,T$	Action $\in \{1..4\}$
$\{b,c,d\}, \{b,d,e\}$	1
$\{c,e\}, \{a,c\}$	2
$\{a\}, \{b,d\}$	3
$\{d\}, \{c\}$	4

An $SLL^1(2)$ parser state is much simpler than a functionally equivalent *SLL* (2) state would be. The $SLL^1(2)$ state is shown in Figure 5.6.

upon $\tau_1 \in \{b,c,d\}$ and $\tau_2 \in \{b,d,e\}$ *predict production* $_1$;
upon $\tau_1 \in \{c,e\}$ and $\tau_2 \in \{a,c\}$ *predict production* $_2$;
upon $\tau_1 \in \{a\}$ and $\tau_2 \in \{b,d\}$ *predict production* $_3$;
upon $\tau_1 \in \{d\}$ and $\tau_2 \in \{c\}$ *predict production* $_4$;

Figure 5.6 State for Example **induces**

A nice way to examine $SLL^1(k)$ decisions is to plot both the real and artificial lookahead vectors. Figure 5.7 is the same as Figure 5.5 except that the artificial vectors have been added; ''×'' is an artificial tuple for action 1 and ''o'' represents an artificial tuple for action 2; there are no artificial tuples for actions 3 and 4.

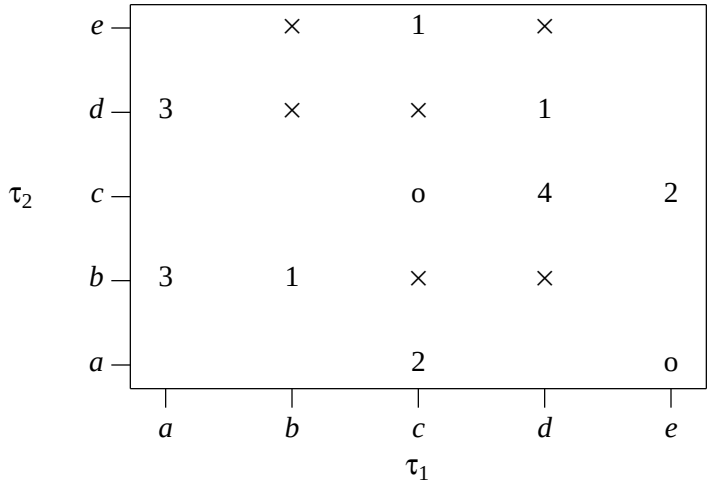


Figure 5.7 Lookahead Vector Plot With Artificial Vectors

None of the real or artificial vectors overlap; hence, the decision is $SLL^1(2)$. A small reduction in strength is sacrificed for a tremendous reduction in space requirements to perform an **induces** relation — from space $O(|T|^k)$ to $O(|T| \times k)$. Moreover, $SLL^1(k)$ can be used to handled the majority of $SLL(k)$ decisions. In the next section, we formalize a number of the observations made up to this point concerning the relative strength of $SLL^1(k)$ and the way in which $SLL^1(k)$ lookahead can be used to separate alternative productions.

5.1.4 $SLL^1(k)$ Formalisms

$SLL^1(k)$ is a very useful language class. It covers most $SLL(k)$ decisions and has linear complexity in terms of grammar analysis and parser decision state complexity. Further, it can be used to reduce the complexity of full $SLL(k)$ analysis and decision state complexity. Chapter 7 describes how $C^1(k)$ decisions in general can be used to reduce the complexity of $C(k)$ decisions. In this section, we define $SLL^1(k)$ decisions formally, show that it is strictly weaker than $SLL(k)$, and show that $SLL^1(k)$ decisions correctly predict productions.

A parsing decision is $SLL(k)$ separable if there does not exist a k-tuple that **induces** more than one parser action (predicts more than one production). We define $SLL^1(k)$ analogously:

Definition: *A decision is* $SLL^1(k)$ *iff there does not exist an artificial (arising from* Λ *set compression) or real lookahead tuple that predicts more than one alternative production.*

An $SLL^1(k)$ grammar is one for which all decisions are $SLL^1(n)$ for some $n \leq k$.

Generic $SLL^1(k)$ decision states are of the form shown in Figure 5.8 where m is the number of productions.

upon $\tau_1 \in \Lambda_1^1$ and ... and $\tau_{n_1} \in \Lambda_{n_1}^1$ *predict production*$_1$
upon $\tau_1 \in \Lambda_1^2$ and ... and $\tau_{n_2} \in \Lambda_{n_2}^2$ *predict production*$_2$
...
upon $\tau_1 \in \Lambda_1^m$ and ... and $\tau_{n_m} \in \Lambda_{n_m}^m$ *predict production*$_m$

Figure 5.8 Generic $SLL^1(k)$ Decision State

where n_i is the maximum lookahead depth required for any production pair involving i. Letting $n_i = k$ renders the generic state the most powerful $SLL^1(k)$ state because it uses all k lookahead depths and compares as most 1-tuples (sets).

The following theorem establishes the relative strength of $SLL^1(k)$.

Theorem 5.1: $SLL^1(k) \subset SLL(k)$ *for* $k > 1$.

Proof:
This is easily shown by example. Consider Grammar G5.2.

$$\begin{aligned} A &\rightarrow B \\ A &\rightarrow ad \\ B &\rightarrow ab \\ B &\rightarrow cd \end{aligned}$$ G5.2

The $SLL^1(2)$ lookahead information, $\{LOOK_1^1\},\{LOOK_2^1\}$, for $A \rightarrow B$ is $\Lambda^1 = \{a,c\},\{b,d\}$ and for $\Lambda^2 = A \rightarrow ad$ is $\{a\},\{d\}$. Clearly, there is no lookahead depth, n, for which $\Lambda_n^1 \bigcap \Lambda_n^2$ is empty. Lookahead *ad* predicts both productions; *a* is in the set of terminals that can be matched at lookahead depth one and *d* is in the set of terminals that can be matched at lookahead depth two for both productions. Grammar G5.2 is non-$SLL^1(2)$, but it is $SLL(2)$ by inspection; this implies that there exists a grammar that is $SLL(k)$, but is not $SLL^1(k)$. Therefore, $SLL^1(k) \subset SLL(k)$.

□

Note that $SLL^1(1)$ and $SLL(1)$ are equally strong as both lookahead computations, $LOOK_1$ and $LOOK_1^1$, compute the same information — the set of terminals recognizable at lookahead depth one.

We turn now to the formalisms needed to test decisions for the $SLL^1(k)$ property.

Lemma 5.1: *A production pair p,q in an* **induces** *relation is* $SLL^1(k) \Leftrightarrow \exists\ n \leq k$ *such that* $\Lambda_n^p \bigcap \Lambda_n^q = \varnothing$.

Proof:

$\Leftarrow$: It is easily seen that $\tau_n \in \Lambda_n^p$ induces p and $\tau_n \in \Lambda_n^q$ induces q if the Λ_n sets are disjoint. This test is $SLL^1(k)$ because, at most, a lookahead depth of $n \leq k$ was employed and only 1-tuple (set) comparisons were done.

$\Rightarrow$: if a production pair is $SLL^1(k)$, it is distinguishable with a lookahead depth of k and with only set memberships. Hence, the most powerful, compliant, distinguishing expression under the $SLL^1(n)$ constraints is the following:

upon $\tau_1 \in \Lambda_1^p$ and ... and $\tau_n \in \Lambda_n^p$ *predict production p*
upon $\tau_1 \in \Lambda_1^q$ and ... and $\tau_n \in \Lambda_n^q$ *predict production q*

Assume the opposite: No Λ_i sets are disjoint for any i=1..k. But, this would imply that the most powerful test cannot distinguish between productions, which contradicts our assumption that it is $SLL^1(k)$. Hence, being $SLL^1(k)$ distinguishable implies that Λ sets for at least one of the lookahead depths, $n \leq k$ are disjoint.

□

Theorem 5.2: *An* **induces** *relation is* $SLL^1(k) \Leftrightarrow \exists\ n \leq k$ *for each production pair p,q such that* $\Lambda_n^p \bigcap \Lambda_n^q = \varnothing$. *If there is only one production in the* **induces** *relation, it is trivially* $SLL^1(k)$ *for* k=0.

Proof:
$\Rightarrow$: if the **induces** relation is $SLL^1(k)$, then any pair of productions p and q must be mutually $SLL^1(n)$ separable for some $n \le k$. By Lemma 5.1, this implies that $\Lambda_n^p \bigcap \Lambda_n^q = \varnothing$.

$\Leftarrow$: If each pair p,q is $SLL^1(n)$ for $n \le k$, then all productions are mutually $SLL^1(k)$ separable and, hence, the **induces** relation must be $SLL^1(k)$.
□

Lemma 5.2: *An* $SLL^1(k)$ *decision uses minimal lookahead depth* $\Leftrightarrow \Lambda_i^p \bigcap \Lambda_i^q \neq \varnothing$ *for* $i = 1..k-1$ *and for all production pairs* p,q.

Proof:
$\Rightarrow$: if an $SLL^1(k)$ decision uses minimal lookahead, then for each production pair p,q there is no $n \le k$ for which the decision is $SLL^1(n)$. Hence, $\Lambda_i^p \bigcap \Lambda_i^q \neq \varnothing$ *for* $i = 1..k-1$.

$\Leftarrow$: If for all production pairs p,q $\Lambda_i^p \bigcap \Lambda_i^q \neq \varnothing$ *for* $i = 1..k-1$, then all production pairs are trivially $SLL^1(k)$ by Theorem 5.2.
□

Minimal lookahead is generally desirable, but there may be a single lookahead depth farther out that separates all production pairs in a decision, which reduces the time and space complexity of a decision state to $O(1)$ and $O(|T|)$, respectively. Sufficient, but not necessary, conditions for a decision to be $SLL^1(k)$ are given in conditions C1a and C1b.

$$\bigcap_{j=1}^{j=m} \Lambda_n^j = \varnothing \qquad \text{C1a}$$

and

$$\bigcap_{j=1}^{j=m} \Lambda_i^j \neq \varnothing \qquad i = 1..n-1 \qquad \text{C1b}$$

for $n \le k$ and where m is the number of productions. These conditions indicate that, for some lookahead depth n, all Λ_n sets are disjoint; hence, this n^{th} set alone is sufficient to deterministically map **induces**. This situation occurs more often than is initially apparent; all $SLL^1(1)$ decisions satisfy these conditions. A decision state of this form is shown in Figure 5.9.

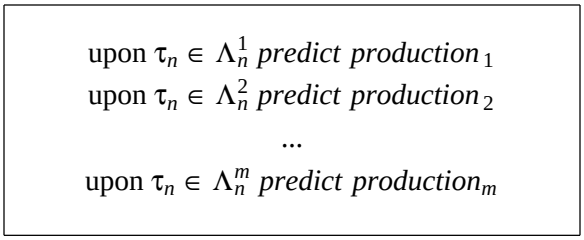


Figure 5.9 Optimized $SLL^1(k)$ **induces** State

Lookahead terminals at depths $1..n-1$ are ignored; each action has at least one lookahead tuple with a terminal appearing at depth $i < n$ that collides with the Λ_i of another production. Surprisingly, this optimization implies that, occasionally, deeper lookahead yields a faster and smaller decision; e.g., while (τ_1, τ_2) might separate productions, τ_3 alone might also. Also recall that, normally, only a subset of the edges emanating from a parser state require $SLL^1(k)$ for $k > 1$. The other edges can be traversed using $SLL^1(1)$. Hence, different decision strategies can be used even within the same state (if a series of lookahead tests are done rather than a single m-ary branch).

In this section, we defined an $SLL^1(k)$ decision to look at most k terminals into the future and to use at most 1-tuple (set) comparisons. $SLL^1(k)$ lookahead information is essentially an approximation to $SLL(k)$. The $SLL^1(k)$ class covers the majority of $SLL(k)$ decisions, has linear grammar analysis complexity, and results in decision states with linear space requirements; full $SLL(k)$ has exponential analysis and space requirements. Although $SLL^1(k)$ is theoretically weaker than $SLL(k)$, it can look farther ahead due to its linearity, which may in practice make it as strong or stronger in some instances.

We presented statistics to support our claim that $SLL^1(k)$ is a useful class of decisions and provided theorems and lemmas that formalize its recognition strength and lead to decision implementation templates. In the next section, we present algorithms that compute $LOOK_k^1$ lookahead information.

5.2 $SLL^1(k)$ Lookahead Computation

Because $SLL^1(k)$ grammars are represented as GLA, lookahead computations can be defined as a collection of simple recurrences. The recurrences specify a walk of the GLA, starting at some state, along which the non-ε edge-labels at distance k are collected into a set called $LOOK_k^1$. This section presents an example $SLL^1(k)$ lookahead computation and provides an algorithm to implement $LOOK_k^1$ following the recurrences given in Section 4.9.1.

5.2.1 Example Lookahead Computation

Before giving algorithms for computing lookahead information, we illustrate how $SLL^1(k)$ $LOOK_k^1$ will behave by constructing lookahead sets for nonterminal A in Grammar G5.3.

$$\begin{aligned} A &\rightarrow Be \\ A &\rightarrow ab \\ B &\rightarrow a \\ B &\rightarrow d \\ C &\rightarrow Bc \end{aligned} \qquad \text{G5.3}$$

Figure 5.10 shows the GLA that would be created from the grammar.

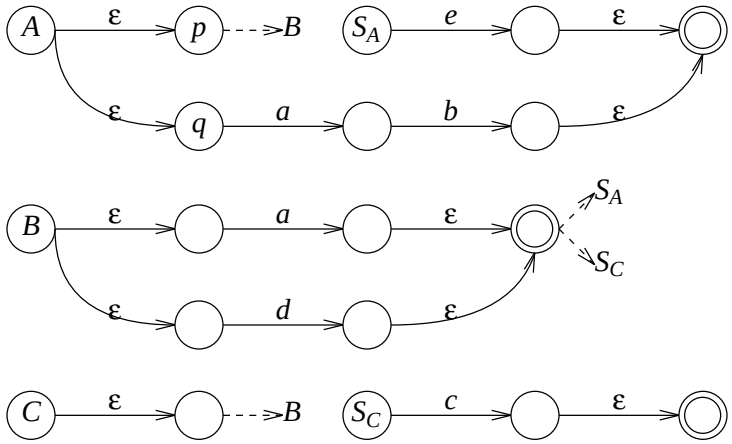


Figure 5.10 Example GLA for *LOOK* Computations

To determine the set of terminal symbols that can possibly occur k terminals in the ‘‘future’’, one simply walks the GLA to find all non-ε-edge labels that appear after having ‘‘walked over’’ $k-1$ edges (ε-edges are traversed in search of other edge types, but are not included in sets or counted as actual terminals); in other words, perform a bounded depth-first search of the GLA. As lower cost decisions are attempted before those with higher cost, $k=1$ is attempted first. $LOOK_1^1(A \rightarrow \bullet Be)$ enters the GLA at node A and traverses the ε-edge to node B whereupon it sees edges a and b at depth one. Hence, $LOOK_1^1(A \rightarrow \bullet Be)$ is $\{a,b\}$. Similarly, $LOOK_1^1(A \rightarrow \bullet ab)$ enters the GLA at A and immediately discovers an edge labeled a at depth one; the lookahead for production two of nonterminal A is therefore $\{a\}$. The **induces** relation for A is tabulated in Table 5.8.

Table 5.8 $SLL^1(1)$ Relation **induces** for Grammar G5.3

Lookahead $\tau_1 \in T$	Action
$\{a,b\}$	*predict* $A \rightarrow Be$
$\{a\}$	*predict* $A \rightarrow ab$

Because *a* **induces** two actions, *A* is not $SLL^1(1)$ (or $SLL(1)$) and $SLL^1(2)$ must be attempted. $LOOK_2^1(A \rightarrow \bullet Be)$ enters the GLA at node *A*, moves past the depth one edges following node *B* and traverses the ε-paths to nodes S_A and S_C looking for $LOOK_1^1(A \rightarrow B \bullet e)$ and $LOOK_1^1(C \rightarrow B \bullet c)$. From S_A, *e* is found at relative depth one; from S_C, *c* is visible at relative depth one. $LOOK_2^1(A \rightarrow \bullet ab)$ moves past the *a* edge following node *A* to find *b* at depth two (relative to *A*). The **induces** relation then becomes deterministic; it is shown in Table 5.9.

Table 5.9 $SLL^1(2)$ **induces** Relation for Grammar G5.3 at k=2

Lookahead $\tau_2 \in T$	Action
$\{c,e\}$	*predict* $A \rightarrow Be$
$\{b\}$	*predict* $A \rightarrow ab$

Table 5.9 gives the minimal information required to induce correct parser action from node *A*. In this way, the phase of a parser generator that generates output parser states does not have to be incredibly clever about how it implements **induces**; the mapping will be optimized heavily by the grammar analysis phase (the number of tuples to map can be made minimal, in general). The associated heterogeneous automaton state is given in Figure 5.11.

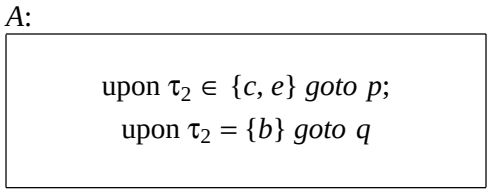


Figure 5.11 Heterogeneous Automaton for Node *A* of Grammar G5.3

This mapping is surprising as it predicts an $SLL(2)$ decision by examining only one lookahead terminal (albeit, the terminal at depth two) — the decision is $SLL^1(2)$.

The example computation in this section illustrates that computing $LOOK_k^1$ is not terribly difficult. The next section provides algorithms that perform the same bounded depth-first-search on the GLA to compute $SLL^1(k)$ lookahead sets.

5.2.2 Algorithms to Compute $SLL^1(k)$ Lookahead

The definitions of $FIRST_k^1$ and $FOLLOW_k^1$ are useful as canonical operations, but $LOOK_k^1$ is computed when building a parser. This section presents two algorithms to compute $LOOK_k^1$ in the $SLL^1(k)$ sense. The first has an exponential complexity, but is straightforward; the second is efficient, but is more complicated. Both algorithms implement the set of $SLL^1(k)$ recurrences as described in Chapter 4.

Given a position in a grammar, $SLL^1(k)$ $LOOK_k^1$ returns the set of terminals that could be matched, while deriving any valid sentence, at a lookahead depth of k. This algorithm operates on GLA constructed as per Section 3.5; it performs what amounts to modified *REACH* and ε-*CLOSURE* operations to find the set of terminals reachable from a given state. Figure 5.12 implements exactly, in pseudo-code, the recurrences in Figure 4.3.

```
function LOOK_k^1( p : Node ) returns set of terminal;
begin
    var rv : set of terminal;

    if p=nil or k=0 then return ∅;
    if p.busy[k] then return ∅;
    p.busy[k] = true;
    if ( p.edge_1 is-a-terminal ) begin
        if ( k>1 ) then rv = LOOK_{k-1}^1(p.edge_1);
        else rv = p.label_1;
    end;
    else rv = LOOK_k^1(p.edge_1);
    rv = rv ∪ LOOK_k^1(p.edge_2);
    p.busy[k] = false;
    return rv;
end LOOK_k^1;
```

Figure 5.12 Inefficient Strong $LOOK_k^1$ Algorithm on GLA

This algorithm is simple for the following reasons:

- The GLA structure used to represent grammars encodes much of the usual procedure used to compute lookahead information. For instance, the *FOLLOW*-links that emanate from each nonterminal GLA accept state encode the fact that the *FOLLOW* of a nonterminal must be included when any production generates fewer than *k* terminals.
- The algorithm computes $LOOK_k^1$ in the $SLL(k)$ sense; i.e. context-insensitive *FOLLOW* sets may be used whereas in $LL(k)$ they cannot; see Chapter 7.
- Results are not saved for use by future computations.
- At most two arcs emanate from any GLA state including nonterminal entry and exit states.
- Computations maintain and return sets of terminals not sets of *k*-tuples.

This version of $LOOK_k^1$ is naive because it does not save results so that future computations do not repeat the work. Because $LOOK_k^1$ is an independent function for each *k*, an obvious improvement is to provide a set of *k* caches for each entry and exit node associated with a nonterminal. In this way, no computation would be performed more than once, which is quite common during grammar analysis. If no computation cycles were possible, caching would be an extremely simple addition to the $LOOK_k^1$ algorithm in Figure 5.12 because computations could not terminate early resulting in incomplete information. Unfortunately, cycles are common and an operation that is incomplete cannot be cached in an obvious way. It would appear that partial results must be cached and that the results must be completed after the cycle completes. In other words, after each $LOOK_k^1$ operation completed, it would correct all cache entries that were incomplete by inserting the necessary terminal set. This mechanism would work, but a simpler, more elegant solution exists.

Recall from Section 4.9.3 that all *LOOK* computations in a cycle result in the same lookahead information. Therefore, computations that must terminate early due to cycle detection can return the partial results of that computation branch and then cache the ‘‘result’’ that that *LOOK* branch is a member of a cycle. Later, when the same *LOOK* computation is requested, its cache entry will point to the computation result for the entire cycle. If the cycle result has not be completed, the cache simply returns a reference to the computation that will eventually be finished.

Cache entries consist of a set of terminals and a completion flag, where the completion flag is true if the cache entry may be used directly; a completion flag of false implies that the entry is a reference to the cache entry of another nonterminal which is the head of a cycle. A cache entry exists for entry and exit state of each nonterminal and for each lookahead depth. The $LOOK_k^1$ computation cache is, therefore, of size $O(|T| \times k \times |N|)$.

Let us re-examine the grammar from Section 4.9.3 as an example of $LOOK_1^1$ cyclic computation caching

$$A \rightarrow a_1 Ba$$
$$A \rightarrow a_2 B$$
$$B \rightarrow a_3 C$$
$$C \rightarrow a_4 A$$
$$C \rightarrow$$

Consider the following computation sequence:

$$LOOK_1^1(B \rightarrow a_3 C \bullet) = LOOK_1^1(A \rightarrow a_2 B \bullet) \cup a$$
$$LOOK_1^1(A \rightarrow a_2 B \bullet) = LOOK_1^1(C \rightarrow a_4 A \bullet)$$
$$LOOK_1^1(C \rightarrow a_4 A \bullet) = LOOK_1^1(B \rightarrow a_3 C \bullet)$$

$LOOK_1^1(B \rightarrow a_3 C \bullet)$ eventually causes the invocation of $LOOK_1^1(C \rightarrow a_4 A \bullet)$ which cannot complete due to its need for $LOOK_1^1(B \rightarrow a_3 C \bullet)$. Therefore, the cache entry for $LOOK_1^1(C \rightarrow a_4 A \bullet)$ is $\{imag(B,FOLLOW)\}$ and labeled as incomplete where $imag(B,FOLLOW)$ is an imaginary terminal representing a cycle to nonterminal A in the *FOLLOW* sense; in practice, only the nonterminal is recorded because when requested again, the requesting function invocation knows which sense of cycle occurred (the cache is connected to a GLA entry or exit node). The computation specifies, via a return parameter, that a cycle to C occurred. $LOOK_1^1(A \rightarrow a_2 B \bullet)$ invokes a computation which results in a cycle; hence, $LOOK_1^1(A \rightarrow a_2 B \bullet)$ is a member of that cycle and also has an incomplete cache entry of $\{imag(B,FOLLOW)\}$. Similarly, $LOOK_1^1(B \rightarrow a_3 C \bullet)$ realizes that it is a member of cycle, but a cycle to itself. The computation can do no more work and considers itself complete. The cache entry for $LOOK_1^1(B \rightarrow a_3 C \bullet)$ is $\{a\}$ and is complete. Future requests for any of the incomplete computations would examine the entry for $LOOK_1^1(B \rightarrow a_3 C \bullet)$, find it complete, complete its own computation from that information, and finally return a copy of its now complete cache entry. Cycles can theoretically occur from traversals of both edges emanating from an GLA state. In this case, the computation is a member of two cycles, which effectively yields a bigger cycle. The cache may refer to either cycle arbitrarily without effecting the result of the computation.

Figures 5.13, 5.14, and 5.14 comprise an efficient algorithm to compute $LOOK_k^1$ using the caching mechanism just described.

```
function LOOK^1_k( p : Node, var cycle : nonterminal ) returns set of terminal;
begin
    var rv : set of terminal;
    var cycle_1, cycle_2 : nonterminal;

    cycle_1 = not-a-cycle;
    cycle_2 = not-a-cycle;
    if ( p is-node-with-cache and p.cache[k] not-empty ) then return retrieve-from-cache( p, cycle );
    if p.busy[k] then begin
        cycle = p.rule;
        return ∅;
    end;
    p.busy[k] = true;
    if ( p.edge_1 is-a-terminal ) then begin
        if ( k >1 ) then rv = LOOK^1_{k-1}(p.edge_1, cycle_1);
        else rv = p.label_1;
        if ( cycle_1 is-cycle-to-current-node ) then cycle_1 = not-a-cycle;
    end;
    else begin
        rv = LOOK^1_k(p.edge_1, cycle_1);
        if ( cycle_1 is-cycle-to-current-node ) then cycle_1 = not-a-cycle;
    end
    rv = rv ∪ LOOK^1_k(p.edge_2, cycle 2);
    if ( cycle_2 is-cycle-to-current-node ) then cycle_2 = not-a-cycle;
    p.busy[k] = false;
    if ( p is-node-with-cache ) then store-into-cache( rv, p, cycle_1, cycle_2 );
    return rv;
end LOOK^1_k;
```

Figure 5.13 Efficient Strong $LOOK^1_k$ Algorithm on GLA

```
function retrieve-from-cache( p : Node, var cycle : nonterminal ) returns set of terminal;
begin
    var nt : nonterminal;
    var node : Node;

    if ( p.cache[k] is-complete ) then return p.cache[k];
    else begin
        nt = imaginary-terminal-to-nonterminal( only-element-of( p.cache[k] ) );
        if ( p is-entry-node ) then node = entry-node-of( nt );
        else node = exit-node-of( nt );
        if ( node.cache[k] is-complete ) then return set-dup( node.cache[k] );
        else begin
            cycle = nt;
            return ∅;
        end;
    end;
end retrieve-from-cache;
```

Figure 5.14 Cache Retrieval for Efficient Strong $LOOK_k^1$

```
procedure store-into-cache(   rv : set of terminal,
                              p : Node,
                              cycle1 : nonterminal,
                              cycle2 : nonterminal );
begin
    var c : nonterminal;

    /* cache this set for use by other functions if complete */
    if ( cycle1, cycle2 are-not-cycles ) then begin
        p.cache[k] = set-dup( rv );
        indicate-complete( p.cache[k] );
        return;
    end

    if ( cycle1 is-cycle or cycle2 is-cycle ) then
    begin
        if ( cycle1 is-cycle ) then c = cycle1;
        else c = cycle2;
        p.cache[k] = set-of( nonterminal-to-imaginary-terminal( c ) );
        indicate-incomplete( p.cache[k] );
    end
end store-into-cache;
```

Figure 5.15 Cache Storage for Efficient Strong $LOOK_k^1$

Once lookahead information has been computed, testing for the $SLL^1(k)$ property is a simple matter of applying the theorems and lemmas in Section 5.11.4.

5.3 Testing for the $SLL^1(k)$ Property

This section provides an algorithm to test a grammar for $SLL^1(k)$ determinism. The advantage of this algorithm is that is has linear time complexity and $SLL^1(k)$ is close to $SLL(k)$ in strength. Figure 5.16 presents a procedure which must be applied to each nonterminal in N.

```
procedure testSLL^1( rule : nonterminal, max_k : integer );
begin
    k = 1;
    p = first-production-of rule;
    while p ≠ nil do begin
        f1 = LOOK_k^1(p.edge_1);
        q = p.edge_2;
        while q ≠ nil do begin
            f2 = LOOK_k^1(q.edge_1);
            while f1 ∩ f2 ≠ ∅ do begin
                if k=max_k then report-nondeterminism;
                else begin
                    k = k+1;
                    f1 = LOOK_k^1(p.edge_1);
                    f2 = LOOK_k^1(q.edge_1);
                end;
            end;
            q = q.edge_2;
        end;
        p = p.edge_2;
    end;
end testSLL^1;
```

Figure 5.16 Algorithm on GLA to Test $SLL^1(k)$ Determinism

The outer two loops iterate through all

$$\binom{m}{2} = \frac{m \times (m-1)}{2}$$

unique production pairs where m is the number of productions for some nonterminal; production pair separability over all production pairs in a decision implies that the entire decision is deterministic. As per section 5.11.4, to guarantee production pair separability, it is sufficient to find a single lookahead depth $n \leq k$ that has no terminals in common. Therefore, only Λ_n sets are examined for each lookahead depth n rather and all 1..n at each iteration. The innermost loop is performed until either a lookahead depth is found that separates the current production pair or the maximum allowable lookahead depth, k, is reached. If the maximum lookahead depth is reached without resolving the production pair prediction problem, the decision is not $SLL^1(k)$ due, at least, to this production pair.

To define the complexity of this algorithm, we separate the cost of computing lookahead information from the cost of iterating over all productions and all nonterminals. As always, we will assume the worst-case scenario. Ignoring lookahead computation costs, for a single nonterminal, *testSLL*^{1} requires space proportional to $|T|$, the size of a set of nonterminals, and time proportional to

$$O(\left\lceil \frac{|P|}{|N|} \right\rceil^2 \times k)$$

where $|P|/|N|$ is the average number of productions per nonterminal (this is a constant less than eight normally in practice). The space required to test all nonterminals is dominated by the space required to compute all $LOOK_k^1$ sets — $O(|G| \times k \times |T|)$. Multiplying for each nonterminal and adding in the time for $LOOK_k^1$ computations established earlier,

$$O(|G| \times k \times |T| + \frac{|P|^2}{|N|} \times k)$$

or, roughly $O(|G| \times k \times |T|)$, is required to test all nonterminals for the $SLL^1(k)$ property. $SLL^1(k)$ testing is therefore a linear function of k for a fixed grammar.

Once all parser decision states have been tested for the $SLL^1(k)$ property, parser construction may begin. The next section describes a simple and effective decision state construction mechanism.

5.4 $SLL^1(k)$ Parser Construction

The construction of *SLL* (1)-based parsers is well understood and sufficiently covered in the literature. In contrast, only theoretical methods exists for lookahead depths greater than one. These techniques are generally simple extensions to those used for *SLL* (1). To construct practical parsers, we describe parsers as heterogeneous automata; each decision state can use a different lookahead expression. Heterogeneous automata can be implemented either as a group of independent states that control the parse (without an interpreter) or as a set of mutually recursive functions or procedures. We choose recursive-descent as the best choice for the implementation of all *LL*-based parsers because of the great flexibility it affords. Although the lookahead decisions themselves are the focus, complete parsers for nonterminals of interest will be constructed.

Once a recursive-descent parser has entered a function matching some nonterminal, a prediction expression, or series of expressions, must indicate which code to execute; i.e. which production to apply. This section suggests one of the possible prediction expression mechanisms — a series of tests, one for each production to predict. This method is the slowest in terms of parsing speed, but is easiest to implement, has the smallest space requirements, and has an acceptably fast average execution time; it has been used in ANTLR [PDC92] to construct *LALL* (*k*) parsers.

In general, $SLL^1(k)$ nonterminal parsing decisions will be of the form shown in Figure 5.17.

procedure *A*;
begin
    if $\tau_1 \in \Lambda_1^1$ **and** $\tau_2 \in \Lambda_2^1$ **and** ... **and** $\tau_n \in \Lambda_n^1$ **then begin**
        match production 1;
    end;
    elseif $\tau_1 \in \Lambda_1^2$ **and** $\tau_2 \in \Lambda_2^2$ **and** ... **and** $\tau_{n'} \in \Lambda_n^{2,}$ **then begin**
        match production 2;
    end;
    ...
    elseif $\tau_1 \in \Lambda_1^m$ **and** $\tau_2 \in \Lambda_2^m$ **and** ... **and** $\tau_{n''} \in \Lambda_n^{m,,}$ **then begin**
        match production m;
    end;
end *A*;

Figure 5.17 $SLL^1(k)$ Nonterminal Decision Template

Terminal references are implemented as MATCH(*a*), a simple macro, which ensures that the current lookahead symbol matches *a* and consumes a terminal. Nonterminal references are merely calls to the appropriate procedure. The lookahead depth, even within one decision, can vary according to the structure of the productions within that grammar decision point; this saves both analysis and parser time and space. From a user-semantics point of view, downward-

inheritance and upward-inheritance are trivially implemented as arguments and return values respectively (recursive-descent parsers have excellent semantic flexibility).

Consider the recognition of Grammar 5.6, which is $SLL^1(3)$.

$$
\begin{aligned}
A &\rightarrow Ba \\
A &\rightarrow Cc \\
A &\rightarrow D \\
A &\rightarrow af \\
B &\rightarrow bc \\
B &\rightarrow eg \\
C &\rightarrow bg \\
C &\rightarrow ef \\
D &\rightarrow st \\
D &\rightarrow ut
\end{aligned}
\qquad \text{G5.6}
$$

Although nonterminal A is $SLL^1(3)$, nonterminals B, C, and D are $SLL^1(1)$ and, hence, their implementations would be much smaller. The **induces** relation of A is shown in Table 5.10.

Table 5.10 Example $SLL^1(3)$ **induces** Relation

Lookahead τ_1, τ_2, $\tau_3 \in T,T,T$	Action
$\Lambda_1^1, \Lambda_2^1, \Lambda_3^1 = \{b,e\}, \{c,g\}, \{a\}$	*predict* $A \rightarrow Ba$
$\Lambda_1^2, \Lambda_2^2, \Lambda_3^2 = \{b,e\}, \{g,f\}, \{c\}$	*predict* $A \rightarrow C$
$\Lambda_1^3, \Lambda_2^3, \Lambda_3^3 = \{s,u\}, \{t\}, \{\$\}$	*predict* $A \rightarrow D$
$\Lambda_1^4, \Lambda_2^4, \Lambda_3^4 = \{a\}, \{f\}, \{\$\}$	*predict* $A \rightarrow af$

A can be implemented by the procedure in Figure 5.18.

```
procedure A;
begin
    if τ1 ∈ {b,e} and τ2 ∈ {c,g} and τ3 = a then begin
        B;
        MATCH(a);
    end;
    elseif τ1 ∈ {b,e} and τ2 ∈ {g,f} and τ3 = c then begin
        C;
        MATCH(c);
    end;
    elseif τ1 ∈ {s,u} and τ2 = t and τ3 = $ then begin
        D;
    end;
    elseif τ1 = a and τ2 = f and τ3 = $ then begin
        MATCH(a);
        MATCH(f);
    end;
end A;
```

Figure 5.18 $SLL^1(3)$ Implementation of A

Many local optimizations can be made. For example, only the first two productions require more than a single terminal of lookahead and lookahead depth two, τ_2, is not needed to predict any production. Therefore, the prediction expressions can be reduced to that of Figure 5.19.

```
procedure A;
begin
    if τ1 ∈ {b,e} and τ3 = a then begin
        B;
        MATCH(a);
    end;
    elseif τ1 ∈ {b,e} and τ3 = c then begin
        C;
        MATCH(c);
    end;
    elseif τ1 ∈ {s,u} then begin
        D;
    end;
    elseif τ1 = a then begin
        MATCH(a);
        MATCH(f);
    end;
end A;
```

Figure 5.19 Optimization of *A*'s Implementation

The τ_1 terms remain because they distinguish the first two productions from the other two productions. Any lookahead depth used to separate any production pair must be included in their production prediction expression.

The set membership test used in the above examples can be implemented in space $O(|T|/wordsize)$ and in time $O(1)$ where *wordsize* is the width in bits of a machine word. Each unique terminal set is expressed by a particular bit position within an array, *setwd*, indexed by the terminal. For example, ''$setwd[\tau_1]$ & 1'' tests τ_1 for membership in the set numbered 1, where ''&'' is the bitwise ''and'' operator. If the bit is one, τ_1 is a member of that set. *setwd* is a table as in Table 5.11.

Table 5.11 Sample Bit Set Implementation — *setwd* Array

setwd index	bits
a	0 0 0 1
b	0 0 1 0
c	0 0 0 1
d	0 0 1 0
e	0 0 1 0

which indicates that *a* and *c* are members of the first set (bit position 0; set membership testing is a bitwise ‘‘and’’ing with 1) and *b*, *d* and *e* are members of the second set (hence we mask with 2. This type of membership operation is $O(1)$, which results in much better execution speed than the equivalent operation on a list representation of a set (which must be searched).

The **induces** relation is computed implicitly by the $SLL^1(k)$ determinism algorithm. Specifically, the results of each $LOOK_k^1$, computed during the course of the algorithm, are stored in $p.look^1[k]$ where p is the node for which the $LOOK_k^1$ was requested and $p.look^1$ is a results buffer in node p. A parser generation pass then trivially walks the GLA for the grammar generating code according to the templates discussed above. Because the $SLL^1(k)$ determinism algorithm computes only as much lookahead as necessary, the resulting parsers use the least possible amount of lookahead; $p.look^1[i]$ will be nonempty for all $i=1..n$ where $n \leq k$ is the minimum lookahead needed to predict that production.

In this chapter we defined a linear approximation to $SLL(k)$ called $SLL^1(k)$. We also provided empirical data that suggests $SLL^1(k)$ handles most $SLL(k)$ decisions and presented algorithms for computing the $SLL^1(k)$ lookahead operator ($LOOK_k^1$) and for testing for the $SLL^1(k)$ condition. Computing all possible $SLL^1(k)$ $LOOK_k^1$ operations can be done in time and space $O(|G| \times k \times |T|)$; because $SLL^1(k)$ testing is dominated by the cost of computing lookahead information and is, therefore, $O(|G| \times k \ |T|)$. A method for constructing $SLL^1(k)$ parsers was also developed; specifically, a recursive-descent procedure was used for each CFG nonterminal that employed a series of up to k set memberships for each of m alternative productions.

$SLL^1(k)$ has linear grammar analysis and lookahead information characteristics, but is strictly weaker than $SLL(k)$. Unfortunately, $SLL(k)$ has exponentially large lookahead information. In the next chapter, we develop $SLL(k)$ parsers fully and demonstrate how $SLL^1(k)$ grammar analysis and decision state construction methods can be used in conjunction with $SLL(k)$ methods to reduce $SLL(k)$ to near-linear performance.

CHAPTER 6 *SLL*(*k*)

SLL(*k*) parsing decisions have lookahead languages that are of size $O(|T|^k)$ in the worst case. Chapter 5 defined an approximation to *SLL*(*k*), denoted $SLL^1(k)$, that reduced the language size to $O(|T| \times k)$ and handled the majority of *SLL*(*k*) decisions. Practical *SLL*(*k*) parsers, are built by employing the minimum amount of lookahead and by using $SLL^1(k)$ decisions when possible. For the few non-$SLL^1(k)$ decisions, a hybrid $SLL^1(k)$/*SLL*(*k*) decision is constructed that typically has near-linear space requirements as is demonstrated in Section 6.18.

This chapter describes how practical *SLL*(*k*) parsers may be constructed. The first section presents an example that illustrates the difference between $SLL^1(k)$ and *SLL*(*k*) decisions. The second and third sections describe *SLL*(*k*) grammar analysis, which includes lookahead information computation and testing for the *SLL*(*k*) property. The final section describes how *SLL*(*k*) grammar analysis, when combined with $SLL^1(k)$ analysis, can be used to build practical *SLL*(*k*) parsers.

6.1 Example *SLL*(*k*) Grammar

The difference between $SLL^1(k)$ and *SLL*(*k*) can best be described by way of an example. Consider Grammar G6.1, which recognizes a few simple C Language declarations. It is *SLL*(3), but not $SLL^1(3)$.

$$
\begin{array}{l}
D \rightarrow TEF \\
D \rightarrow swS \\
T \rightarrow i \\
T \rightarrow st \\
E \rightarrow wlr \\
E \rightarrow w \\
S \rightarrow b\ struct\_body\ e \\
F \rightarrow b\ function\_body\ e
\end{array}
\qquad \text{G6.1}
$$

where *l* and *r* are left and right parenthesis; *b* and *e* are beginning and ending curly braces ‘‘{, }’’. Nonterminal *D* abstracts a declaration, *T* is a type, *E* is a declarator expression, *S* is a structure body, and *F* is a function body. Terminal *s* represents `struct`, *w* is a word, *i* is `int`, and *t* is a type name. *D* will match sentences such as

s w b struct_body e
s t w
i w b function_body e

which, represented in C, would be

```
struct word { struct_body }
struct structname word
int word { function_body }
```

The associated *SLL* (3) **induces** relation for nonterminal *D* is given in Table 6.1.

Table 6.1 *SLL* (3) **induces** Relation for Nonterminal *D* in Grammar G6.1

Lookahead $(\tau_1,\tau_2,\tau_3) \in T^3$	Action
(i,w,l)	*predict* $D \rightarrow TEF$
(i,w,b)	*predict* $D \rightarrow TEF$
(s,t,w)	*predict* $D \rightarrow TEF$
(s,w,b)	*predict* $D \rightarrow swS$

The results of $SLL^1(3)$ analysis, on the other hand, yields the **induces** relation in Table 6.2.

Table 6.2 $SLL^1(3)$ **induces** Relation for Nonterminal *D* in Grammar G6.1

Lookahead $\tau_1, \tau_2, \tau_3 \in T,T,T$	Action
$\{i,s\},\{w,t\},\{l,b,w\}$	*predict* $D \rightarrow TEF$
$\{s\},\{w\},\{b\}$	*predict* $D \rightarrow swS$

Clearly, there is no lookahead depth, *n*, such that the Λ_n^1 and Λ_n^2 are disjoint; hence, the decision is not $SLL^1(3)$. One is not left with the prospect of testing $O(|T|^k)$ *k*-tuples, however. Combining $SLL^1(k)$ with a few *k*-tuple comparisons can be more efficient than the straightforward approach of huge tables, gigantic DFA's or long sequences of *k*-tuple comparisons. Section 6.18 discusses this at length, but we present two sample *SLL* (*k*) decision states to illustrate the mechanism; see Figures 6.1 and 6.2.

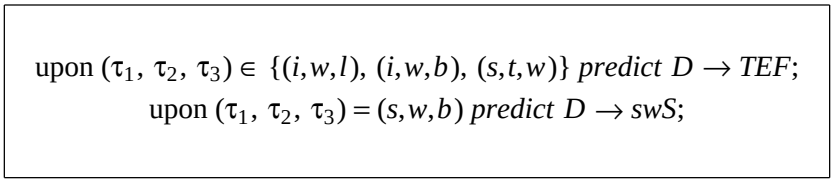


Figure 6.1 Conventional State for Nonterminal *D* in Grammar G6.1

The conventional state, which performs *k*-tuple comparisons, is straightforward, but can quickly become exponential. Even though $SLL^1(3)$ is insufficient for this decision, it is mostly sufficient; the only problem with an $SLL^1(k)$ decision here is that an artificial 3-tuple of $D \rightarrow TEF$, (s,w,b), is a real tuple predicting $D \rightarrow swS$. If one were to use the $SLL^1(k)$ decision template and then tested for this one ‘‘irregularity’’, a correct prediction expression would result. Figure 6.2 illustrates this hybrid approach.

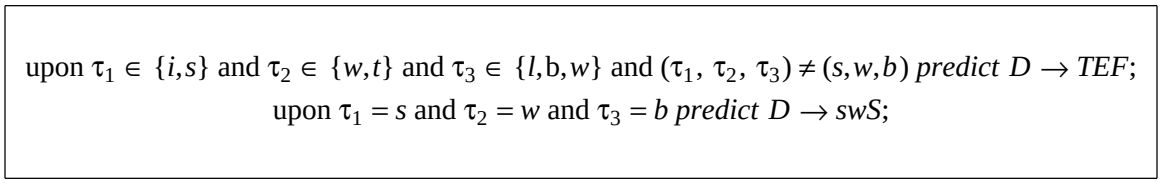


Figure 6.2 Hybrid State for Nonterminal *D* in Grammar G6.1

As a further reduction, consider an implementation of heterogeneous parser states that tests the ‘‘upon’’ expressions sequentially in the order specified rather than doing an *m*-ary branch. In this case, if the second prediction expression appeared first, then the extra tuple comparison on the $D \rightarrow TEF$ predictor is unnecessary. Even though both prediction expressions would match (s,w,b), the first production would be predicted by default, thus, rendering a valid parse.

Because this example is so small, the hybrid $SLL^1(k)/SLL(k)$ state does not appear to be much of a win, but for real examples, *k* set comparisons plus a few *k*-tuple comparisons is much better than $|T|^k$ tuple comparisons or trying to store that many tuples into a hash table.

6.2 *SLL* (*k*) Lookahead Computation

$SLL^1(k)$ analysis returned a set of terminals at depth k away from the initial GLA state. This information is used by $SLL^1(k)$ to make extremely efficient decisions. However, when this type of decision results in a nondeterministic parser state, an *SLL* (*k*) decision must be attempted; i.e. $LOOK_k$ information must be computed.

The GLA representation of a grammar is designed to allow lookahead computations to be described as simple recurrences such as those of Section 4.9.1. The *SLL* (*k*) lookahead language for a grammar position p, $LOOK_k(p)$, is the collection of non-ε-edges along the walks of length k emanating from the GLA state associated with position p. The lookahead information is formed into child-sibling trees, which are much easier to manipulate than DFA's (an equally valid lookahead information representation).

While lookahead computations are easy to define on a GLA, the time and space complexity of an actual computation may be extremely high. There are two sources of nonlinearity in grammar analysis. One source arises from the recursive nature of grammars (see Section 4.9.3 on computation cycles) and another from the exponential size of the lookahead information. This section provides three algorithms that implement the lookahead recurrences of Section 4.9.1: a straightforward algorithm, an algorithm that uses the results of $SLL^1(k)$ analysis to prune the number of walks, and finally an algorithm that caches all results for use by later computations. We begin by illustrating the straightforward computation of lookahead via an example.

6.2.1 Example Lookahead Computation

To illustrate the computation of $LOOK_k$ and the use of trees to compute lookahead information, consider Grammar G6.2.

$$\begin{aligned} A &\rightarrow Be \\ A &\rightarrow ab \\ B &\rightarrow a \\ B &\rightarrow d \\ C &\rightarrow Bc \end{aligned} \qquad \text{G6.2}$$

Figure 5.10 shows the GLA that would be created.

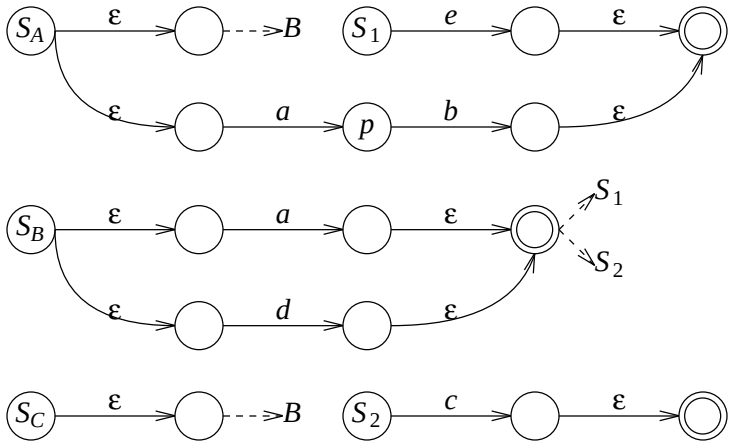


Figure 6.3 Example GLA for *LOOK* Computations

Whereas $LOOK_k^1$ walks an GLA collecting non-ε edge labels into a set, $LOOK_k$ records the paths as well as the non-ε edge-labels. $LOOK_2(A \rightarrow \bullet Be)$ jumps the ε-edge to state *B* where it sees two alternatives. Arbitrarily choosing to traverse edge *a*, $LOOK_2$ finds its first usable piece of information and creates a tree node. Because a terminal edge has been traversed, the lookahead depth has been reduced by one. Therefore, $LOOK_1$ is attempted for the state pointed to by the *a* edge in production one of *B*. $LOOK_1$ traverses both ''*FOLLOW*'' links, finding edges *e* in nonterminal *A* and *c* in nonterminal *C*. Because these can be viewed as an alternative set of terminals, the tree nodes are connected as siblings $e \rightarrow c$. To combine the $LOOK_1$ tree with the previous work, done by $LOOK_2$, the *a* tree node becomes the parent of *e* and *c*:

$$\begin{array}{l} a \\ \downarrow \\ e \rightarrow c \end{array}$$

$LOOK_2$ is not finished yet as it must walk the second alternative of *B*. $LOOK_1$ is applied to the node pointed to by the *d* edge in *B*, which returns $e \rightarrow c$. Edge *d* precedes the $LOOK_1$ operation and hence, the lookahead tree for the second production of *B* is:

$$\begin{array}{l} d \\ \downarrow \\ e \rightarrow c \end{array}$$

The two subtrees with *a* and *d* as roots are alternatives in relation to each other and become siblings in a new, larger tree, which is returned as the lookahead tree for *B*:

$$\begin{array}{lll} a & - - \rightarrow & d \\ \downarrow & & \downarrow \\ e \rightarrow c & & e \rightarrow c \end{array}$$

Computing $LOOK_2(A \rightarrow \bullet ab)$ is straightforwardly:

$$\begin{array}{c} a \\ \downarrow \\ b \end{array}$$

6.2.2 Straightforward $LOOK_k$ Algorithm

The recurrences of Section 4.9.1 can be implemented in a fairly straightforward, but inefficient, manner. Figure 6.4 provides such an algorithm to compute $LOOK_k$.

```
function LOOK_k( p : Node ) returns tree of terminal;
begin
    var t,u : tree of terminal;

    if p=nil or k=0 then return nil;
    if p.busy[k] then return nil;
    p.busy[k] = true;
    if ( p.edge_1 is-a-terminal )
                    p.label_1
        t =             ↓           ; /* create tree with label as root */
            LOOK_{k-1}(p.edge_1)
    else
        t = LOOK_k(p.edge_1);
    u = LOOK_k(p.edge_2);
    p.busy[k] = false;
    if t=nil then return u;
    else return t → u; /* create tree with t,u as siblings */
end LOOK_k;
```

Figure 6.4 Straightforward $SLL(k)$ $LOOK_k$ Algorithm on GLA

The straightforward algorithm makes no attempt to save the results of its computations for future use and, hence, may exhibit exponential behavior derived from the recursive nature of grammars. It most certainly will encounter the exponential size of $LOOK_k$ information. This algorithm is roughly $O(|G|^k \times |T|^k)$ in the worst case. The next section presents an algorithm that, using $SLL^1(k)$ analysis results, attempts to reduce the exponentiality derived from the huge lookahead information.

6.2.3 Constrained $LOOK_k$ Algorithm

This section provides a $LOOK_k$ algorithm that is typically more efficient than the straightforward algorithm, but still does not cache computations; hence, the grammar-derived exponentiality is ignored. It walks the GLA as before except that it does not follow every possible edge. The size of the lookahead information, therefore, is reduced, but the results cannot be cached as they are incomplete. This method is employed by ANTLR, the parser generator in PCCTS [PDC92], to practically generate, without caching, $LALL(k)$ parsers; the worst-case behavior of the constrained algorithm is identical to the best-case/worst-case behavior of the straightforward algorithm. We begin by discussing how the constrained $LOOK_k$ algorithm is used.

Before testing a production pair p,q for the $SLL(k)$ condition, it is tested for the $SLL^1(k)$ condition because $LOOK_k^1$ has linear time and space complexity. When no lookahead depth separates p and q, $\Lambda_n^p \bigcap \Lambda_n^q \neq \varnothing$ for all $1 \leq n \leq k$, the pair is not $SLL^1(k)$ and $SLL(k)$ must be attempted; recall that the Λ_n sets are computed by $LOOK_n^1$ computation. The information gained from $SLL^1(k)$ analysis is still of value. During $SLL(k)$ analysis, rather than compute the set of k-sequences that predict productions p and q, it is sufficient to search for the set of k-sequences that are possibly in common; i.e. in the intersection $\Lambda' = \Lambda_n^p \bigcap \Lambda_n^q \forall n$. Λ' covers the set of all possible k-sequences that could render the pair non-$SLL(k)$-separable. By constraining the $LOOK_k$ computation to traverse only those edges that may lead to a common k-sequence, the typical complexity of the computation can be reduced significantly. The ‘‘constrained’’ algorithm is given in Figure 6.5.

```
function LOOK_k( p : Node, Λ : array of sets ) returns tree of terminal;
begin
    var t,u : tree of terminal;

    if p=nil or k=0 then return nil;
    if p.busy[k] then return nil;
    p.busy[k] = true;
    if p.edge_1 is-a-terminal then begin
        if p.edge_1 ∈ Λ_k then
                        p.label_1
            t =          ↓          ; /* create tree with label as root */
                 LOOK_{k-1}(p.edge_1)
        else
            t = nil;
    end;
    else
        t = LOOK_k(p.edge_1);
    u = LOOK_k(p.edge_2);
    p.busy[k] = false;
    if t=nil then return u;
    else return t → u; /* create tree with t,u as siblings */
end LOOK_k;
```

Figure 6.5 Constrained $SLL(k)$ $LOOK_k$ Algorithm on GLA

The input parameter Λ is the result of $SLL^1(k)$ analysis and is an array comprised of Λ_n for $1 \le n \le k$.

This constrained algorithm attempts to reduce the exponential behavior of the straightforward algorithm by pruning the size of the lookahead information, but, while the typical behavior of this algorithm is good, it is still exponentially complex in the worst case; to compute all $LOOK_k$ trees using the constrained algorithm, the worst-case time and space complexity is $O(|G|^k \times |T|^k)$ as per Section 4.10. The next section describes an algorithm that overcomes the grammar-derived exponentiality, but still must contend with exponential lookahead information size.

6.2.4 $LOOK_k$ Algorithm With Caching

The straightforward algorithm ignores both the grammar-derived and lookahead information size exponentialities while the constrained algorithm attempts to reduce the lookahead information size. In this section, we present an algorithm that reduces $LOOK_k$ complexity by attacking the grammar-derived exponentiality; results of its computations are cached for use by future

computations. A caching mechanism similar to that used by the $LOOK_k^1$ algorithm can be used for this task. However, $LOOK_k^1$ only had to store sets, but $LOOK_k$ has to save trees. Because each decision can have $O(|T|^k)$ lookahead k-tuples, this cache is extremely large in the worst-case. Fortunately, this worst case rarely appears for real grammars.

Cache entries consist of a tree of terminals and a completion flag, where the completion flag is true if the cache entry may be used directly; a completion flag of false implies that the entry is a reference to the cache entry of another nonterminal which is the head of a cycle. A cache entry exists for entry and exit state of each nonterminal and for each lookahead depth. The $LOOK_k$ computation cache size in the worst case is, therefore, of size

$$O(\sum_{i=2}^{k} |T|^i \times |N|)$$

where the $i=1$ case is handled more efficiently by the $LOOK_1^1$ algorithm.

Figures 6.6, 5.14, and 5.14 comprise the caching $LOOK_k$ algorithm; Figure 6.6 implements the GLA recurrences given in Section 4.9.1 and is very similar to the cached $LOOK_k^1$ algorithm in terms of the caching steps. The only real difference lies in the computation of the lookahead information itself.

```
function LOOK_k( p : Node, var cycle : nonterminal ) returns tree of terminal;
begin
    var r,t,u : tree of terminal;
    var cycle_1, cycle_2 : nonterminal;

    cycle_1 = not-a-cycle;
    cycle_2 = not-a-cycle;
    if ( p is-node-with-cache ) then return retrieve-from-cache( p, cycle );
    if p.busy[k] then begin
        cycle = p.rule;
        return nil;
    end;
    p.busy[k] = true;
    if p.edge_1 is-a-terminal then begin
                    p.label_1
        t =             ↓           ; /* create tree with label as root */
            LOOK_{k-1}(p.edge_1, cycle_1)
        if ( cycle_1 is-cycle-to-current-node ) then cycle_1 = not-a-cycle;
    end;
    else begin
        t = LOOK_k(p.edge_1, cycle_1);
        if ( cycle_1 is-cycle-to-current-node ) then cycle_1 = not-a-cycle;
    end
    u = LOOK_k(p.edge_2, cycle_2);
    if ( cycle_2 is-cycle-to-current-node ) then cycle_2 = not-a-cycle;
    p.busy[k] = false;
    if t=nil then r = u;
    else r = t → u;  /* create tree with t,u as siblings */
    if ( p is-node-with-cache ) then store-into-cache( r, p, cycle_1, cycle_2 );
    return r;
end LOOK_k;
```

Figure 6.6 $LOOK_k$ Algorithm on GLA with Caching

Retrieving trees from the cache is done by searching for a complete cache entry in one of three ways. First, if the cache for a node, *p*, is nonempty and has a complete cache, return that tree. If the cache is not complete, but is nonempty, the cache contains a single tree node which points the nonterminal at the head of the cycle with *p* as a member; the cache entry for that nonterminal at the head of the cycle may be used (if it is complete). Thirdly, if the cache for *p* is empty, cache entries at larger values of *k* are sought as they contain a superset of the $LOOK_k$ information; in general, the first $n \leq k$ levels from the root of a $LOOK_k$ tree represents $LOOK_n$. If no complete cache entry is found, the actual computation is attempted.

```
function retrieve-from-cache( p : Node, var cycle : nonterminal  ) returns tree of terminal;
begin
        var nt : nonterminal;
        var node : Node;

        if p.cache [k ] not-empty then begin
                if ( p.cache [k ] is-complete ) then return deep-dup-unique( p.cache [k ] );
                else begin
                        nt = imaginary-terminal-to-nonterminal( p.cache [k ] );
                        if ( p is-entry-node ) then node = entry-node-of( nt );
                        else node = exit-node-of( nt );
                        if ( node.cache [k ] is-complete ) then
                                return deep-dup-unique( node.cache [k ] );
                        else begin
                                cycle = nt;
                                return nil;
                        end;
                end;
        end;
        else begin   /* look for cache entries at higher values of k */
                for i = k+1 to maximum-k-in-cache do begin
                        if ( p.cache [k ] not-empty and p.cache [i ] is-complete ) then
                                return bounded-deep-dup-unique( p.cache [i ], k );
                end;
        end;
end retrieve-from-cache;
```

Figure 6.7 Cache Retrieval for Efficient $SLL(k)$ $LOOK_k$

When a tree is stored into the cache, the $LOOK_k$ algorithm returns only a reference to that entry (a single-node tree which points to the cache entry) rather than making a complete copy of the exponentially large tree. This allows a small amount of tree compression akin to common subexpression elimination as future computations will have references to a cache entry rather than a copy of the full tree. In most situations, however, deep, unique copies are made (reference nodes are expanded to be copies of the trees referenced and terminals occur at most once at a k level in the tree). $LOOK_k$ is very fast when shallow, versus deep, copies are made because tree are of size $O(|T|^k)$, but, testing for $C(k)$-determinism is much slower due to the fact that it must traverse many reference nodes before it discovers an actual terminal node. It is a tradeoff that appears to favor making deep copies to save time later during nondeterminism detection.

```
procedure store-into-cache(   var r : tree of terminal,
                              p : Node,
                              cycle1 : nonterminal,
                              cycle2 : nonterminal );
begin
    var c : nonterminal;

    /* cache this set for use by other functions if complete */
    if ( cycle1, cycle2 are-not-cycles ) then begin
        p.cache[k] = r;
        r = reference-to(r);
        indicate-complete( p.cache[k] );
        return;
    end

    /* if part of cycle, cache only cycle tree entry pointing to cycle nonterminal */
    if ( cycle1 is-cycle or cycle2 is-cycle ) then
    begin
        if ( cycle1 is-cycle ) then c = cycle1;
        else c = cycle2;
        p.cache[k] = tree-node( nonterminal-to-imaginary-terminal( c ) );
        indicate-incomplete( p.cache[k] );
    end
end store-into-cache;
```

Figure 6.8 Cache Storage for Efficient $SLL(k)$ $LOOK_k$

The cached algorithm avoids the exponentiality derived from recursive grammars, which cause redundant computations, but still encounters huge lookahead information trees. To compute all $LOOK_k$ trees, the worst-case time and space complexity is $O(|G| \times k \times |T|^k)$ as per Section 4.10.

This section provided an example $LOOK_k$ computation that demonstrated how GLA's are traversed and how child-sibling trees are employed to store lookahead information. Three algorithms were then presented that implement the recurrences in Section 4.9.1. The straightforward algorithm has two exponential terms in its complexity: $|G|^k$ implies that the recursive nature of grammars forces redundant computations and $|T|^k$ reflects the worst-case size of lookahead information. The constrained algorithm reduced the typical size of the lookahead information by constraining its walk of the GLA. The caching algorithm attacked the grammar-derived exponentiality by saving the results of computations. The straightforward and constrained lookahead computation algorithms require $O(|G|^k \times |T|^k)$ time and space complexity in the worst case; the constrained algorithm may encounter situations in which the lookahead cannot be constrained and, hence, has the same worst-case complexity as the straightforward algorithm. The caching algorithm has the best time and space complexity of the three at $O(|G| \times k \times |T|^k)$. The $|G|^k$ term has been reduced to $|G| \times k$ because of computation caching.

6.3 Testing for the $SLL(k)$ Property

Testing a grammar for the $SLL(k)$ property is very similar to testing for the $SLL^1(k)$ as described in Section 5.13. As before, our approach involves computing $SLL(k)$ lookahead sets and then testing each grammar decision point for determinism. Previous work in this area did not compute lookahead sets and is, hence, less practical when parser generation is the goal of grammar analysis. As always great care is taken to avoid the exponentiality of lookahead information.

Recall the hierarchical approach of reducing lookahead computation complexity used by algorithms in this thesis. First, only the lookahead sets that are needed to parse the language described by the grammar are computed. Second, the lookahead depth, k, is modulated to use minimum lookahead. Third, $SLL^1(k)$ computations, which are linear in k, are attempted before full $SLL(k)$ computations. Algorithms which examine lookahead information, such as the $SLL(k)$ determinism algorithm, request lookahead computations in this manner. This section presents statistics concerning the relative efficiency (run time) of analyzing grammars and provides an algorithm for testing grammars for the $SLL(k)$ property.

6.3.1 Characteristics of *SLL* (*k*) Determinism

This section presents statistics regarding the efficiency of testing grammars for the $SLL^1(k)$ condition and the *SLL* (*k*) condition. The *SLL* (*k*) algorithm uses a combination of $LOOK_k^1$ and $LOOK_k$ computations.

The time complexity of testing for the *SLL* (*k*) property can be viewed as a function of the number of *LOOK* operations and the amount of lookahead information which must be examined for determinism. The caching mechanisms employed by our efficient grammar analysis algorithms bound the number of *LOOK* operations on a state *p* to $|G| \times k$ — there are only *k* operations that are defined on each state and the results are cached for use by later requests for that information. Hence, empirical studies should show that the average number of *LOOK* requests per decision state is a linear function of *k*. Figure 6.9 shows that, indeed, when the number of *LOOK* operations per decision state was averaged over 22 sample grammars, the number of *LOOK* operations is a linear function of *k*.

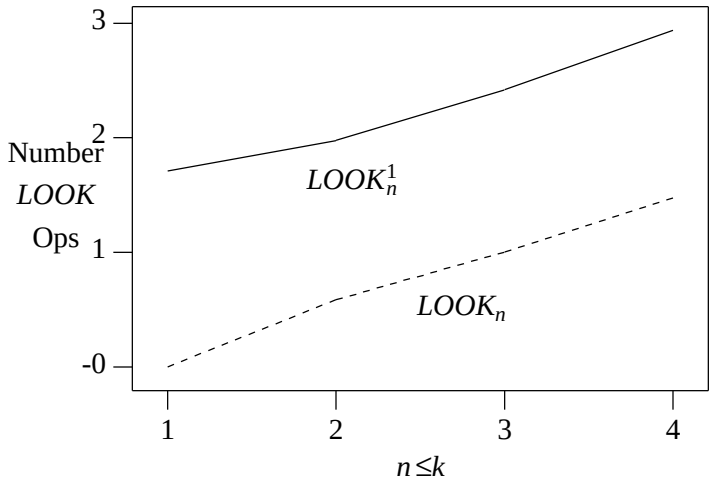


Figure 6.9 Average Number of *LOOK* Operations per Decision for *SLL* (*n*) Determinism

Notice, for $k=1$, $LOOK_n$ is never attempted because $LOOK_1^1$ is equivalent to $LOOK_1$ (both compute sets of terminals), but $LOOK_1^1$ is more efficient. The $LOOK_n$ curve follows roughly the function $k/4$, which suggests that, because most decisions are *SLL* (1), increasing the maximum allowable *k* does not dramatically increase the number of *LOOK* requests at the maximum depth. Nondeterministic decisions force the *SLL* (*k*) determinism algorithm to consider deeper and deeper lookahead since it does not know whether the decisions is nondeterministic for any *k* or merely that an insufficient amount of lookahead has been attempted.

The number of *LOOK* computations is a linear function of k, but the cost of each can be exponential. Hence, the time needed to analyze a grammar for the $SLL(k)$ property is exponential for $k>1$. On the other hand, testing a grammar for the $SLL^1(k)$ condition (a covering approximation to $SLL(k)$), is a linear problem. To strengthen our claim of practicality, we present Figure 6.10 that illustrates the amount of time needed to analyze $SLL(k)$ grammars. We have included the time for ANTLR (ANother Tool for Language Recognition) [PDC92], the $LALL(k)$ parser generator of PCCTS, to analyze the same sample grammars; ANTLR is a widely used and considered practical, thus, providing a good benchmark. The $SLL(k)$ algorithm uses a combination of $LOOK_k^1$ and $LOOK_k$ computations. Figure 6.10 demonstrates that $SLL^1(k)$ is much more efficient and is roughly linear whereas the $SLL(k)$ analysis is exponential in k; $SLL(3)$ is still, however, very practical. The times were averaged over 22 sample grammars.

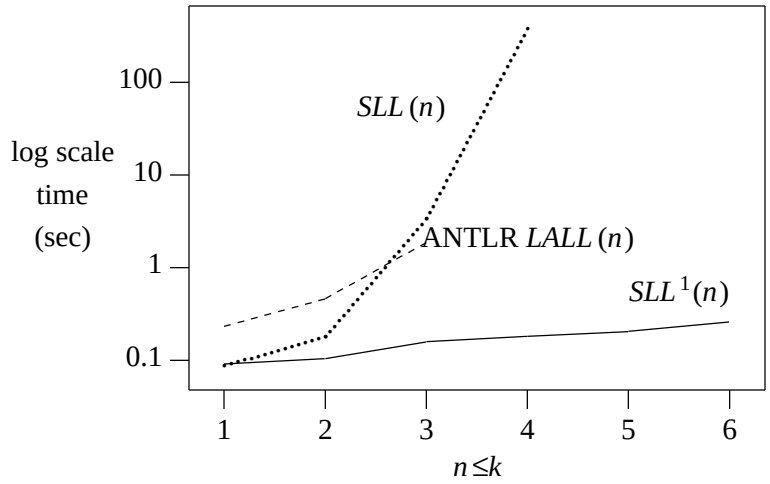


Figure 6.10 Average Analysis Time for $SLL^1(n)$, $SLL(n)$, and $LALL(n)$ Determinism

The ANTLR analysis curve does not include sample 14 as ANTLR could not compute $LALL(2)$ or $LALL(3)$ for this grammar (program exceeded internal data structure constraints); also, ANTLR does not terminate for $LALL(4)$ for most of the grammars. The figure indicates that constrained $LOOK_k$ is often very fast, but the grammar-derived nonlinearity becomes significant at $k=4$; the cached $LOOK_k$ is exponential not in the grammar, but in the size of the lookahead trees whereas the constrained algorithm manipulates pruned trees.

This section demonstrated that the number of lookahead operations per decision state can be reduced from an exponential to a linear function with computation caching. Unfortunately, each computation can be exponentially large, but for small k, we have shown that $C(k)$ parsing is still practical.

6.3.2 Algorithm for Testing for the $SLL(k)$ Property

Avoiding full $LOOK_k$ operations is the primary goal of grammar analysis. Since $SLL^1(k) \subset SLL(k)$, if a lookahead decision is $SLL^1(k)$ it is also $SLL(k)$ and need not be tested for the $SLL(k)$ property; hence, $LOOK_k^1$ operations are attempted first. When $SLL^1(k)$ is insufficient, $LOOK_k$ operations must be computed. The algorithm to test a grammar for the $SLL(k)$ property is presented in Figure 6.11. It is identical to the algorithm for $SLL^1(k)$ except that, upon reaching the maximum lookahead depth without having separated the production pair, full $SLL(k)$ determinism is examined, whereas, the $SLL^1(k)$ algorithm would report a nondeterminism at this point.

```
procedure testSLL( rule : nonterminal, max_k : integer );
begin
    k = 1;
    p = first-production-of rule;
    while p ≠ nil do begin
        f1 = LOOK_k^1(p.edge1);
        q = p.edge2;
        while q ≠ nil do begin
            f2 = LOOK_k^1(q.edge1);
            while f1 ⋂ f2 ≠ ∅ do begin
                Λ = f1 ⋂ f2;
                if k=max_k then testFullLL (p,q,k, Λ);
                else begin
                    k = k+1;
                    f1 = LOOK_k^1(p.edge1);
                    f2 = LOOK_k^1(q.edge1);
                end;
            end;
            q = q.edge2;
        end;
        p = p.edge2;
    end;
end testSLL;
```

Figure 6.11 Algorithm on GLA to Test *SLL* (*k*) Determinism

Only the non-$SLL^1(k)$ production pairs in a decision point are examined for $SLL(k)$ property; if the other pairs are $SLL^1(k)$ they are trivially $SLL(k)$. Although it is not performed by this algorithm, when $|\Lambda_i| = 1$ for $i=1..k-1$, no artificial tuples are generated by the Λ compression and, hence, computing full $SLL(k)$ information is unnecessary.

Just as the results of $SLL^1(k)$ analysis can be used to reduce the complexity of $SLL(k)$ analysis (see Section 6.16.3 on the constrained $LOOK_k$ algorithm), $SLL^1(k)$ lookahead information can be used to reduce the complexity of testing two productions for the $SLL(k)$ property. The Λ_i sets, which are the intersection of $LOOK_i^1(p)$ and $LOOK_i^1(q)$, are computed for each lookahead depth i; the information is used by the *testFullSLL* procedure to reduce the amount of time required to test for the $SLL(k)$ property. The algorithm to test a production pair for the $SLL(k)$ property is given in Figure 6.12.

```
procedure testFullSLL( p,q : Node, k : integer, Λ : array of sets );
begin
    if k >1 then begin
        f1 = LOOKk(p.edge1);
        f2 = LOOKk(q.edge1);
    end;
    for n = 2 to k do begin
        t = permutation( Λ, n );
        while t ≠ nil do begin
            if tree-member( f1, t, n ) and tree-member( f2, t, n ) and n=k then
                report-nondeterminism;
            t = permutation( Λ, n );
        end;
    end;
end testFullSLL;
```

Figure 6.12 Algorithm on GLA to Test for $SLL(k)$ Determinism

The function permutation(Λ,n) used by *testFullSLL* returns a new lookahead n-string (tree of depth n with n elements) from the lookahead space covered by Λ_1 ... Λ_n. The tree-member(f,t,n) function returns true if lookahead string t is a member of tree f up to a depth of n else it returns false.

Because $SLL^1(1)$ is equivalent to $SLL(1)$, nothing is done by *testFullSLL* when $k=1$. For $k>1$, $LOOK_k$ is generally requested for both productions of the production pair. Testing for the $SLL(k)$ property is a simple matter of testing all permutations of length n in the Λ sets against the first n levels in the lookahead trees. Only those n-tuples appearing the the Λ sets are attempted as the Λ sets cover the intersection of the lookahead trees for productions p and q, thus eliminating a large number of unnecessary comparisons. Any lookahead n-tuple that p and q have in common is covered by the Λ set.

6.3.3 Complexity of Testing for the *SLL* (k) Property

The space complexity of *testSLL* is the same as *testSLL*^{1}, but with the additional space required for *testFullSLL*, which in turn requires space for $LOOK_k$ computations. Space complexity of *testSLL* is, therefore, dominated by that of computing $LOOK_k^1$ and $LOOK_k$ information, which totals to $O(|G| \times k \times |T| + |G| \times k \times |T|^k)$ as per Section 4.10, or simply $O(|G| \times k \times |T|^k)$.

To establish the worst-case time complexity of *testSLL*, we recall that the time complexity of *testSLL*^{1} is $O(|G| \times k \times |T|)$ which will be the same for *testSLL* without the time needed to perform *testFullSLL*. The time required to compute all full *SLL* (k) lookahead information is $O(|G| \times k \times |T|^k)$ as per Section 4.10. Turning to the examination of the lookahead, we observe that the outer loop of *testFullSLL* performs $k-1$ iterations and tests permutations of Λ, looking for the minimum k which separates the two productions. The number of permutations possible for Λ_k is $|T|^k$ in the worst case ($\Lambda_i \subseteq T$); hence, the inner loop could iterate $O(|T|^k)$ times. Each permutation of length n is tested against the lookahead trees for p and q, which requires time proportional to the tree size; this could be reduced to $O(n)$ by representing the trees as DFA's for determinism testing purposes. Time to perform the lookahead examination portion of *testFullSLL* for one production pair is then $O(|T|^{k+1} \times k)$. In the worst case, each nonterminal and each production pair of the nonterminal, needs *testFullSLL* yielding a time complexity, including the cost of lookahead computation, of

$$O\left(|G| \times k \times |T|^k + |N| \times \left(\frac{|P|}{|N|}\right)^2 \times |T|^{k+1} \times k\right)$$

where

$$\left(\frac{|P|}{|N|}\right)^2$$

is the average number of production pairs per nonterminal ($|P|/|N|$ is a constant less than eight normally in practice). Therefore, time complexity for all $|N|$ invocations of *testSLL* plus the time for all possible invocations of *testFullSLL* is

$$O\left(|G| \times k \times |T| + |G| \times k \times |T|^k + \frac{|P|^2}{|N|} \times |T|^{k+1} \times k\right)$$

Simplifying, we obtain

$$O(|G| \times k \times |T|^k + \frac{|P|^2}{|N|} \times |T|^{k+1} \times k)$$

which is roughly $O(|G| \times k \times |T|^{k+1})$. By improving the strategy by which we compare k-strings against lookahead trees, the complexity can be reduced to $O((|G| + |P|^2/|N|) \times k \times |T|^k)$.

This section explored a method for testing grammars for the $SLL(k)$ property. Our approach resolves as many decisions as possible with $SLL^1(k)$ lookahead, but failing that, employs full $SLL(k)$. In either case, the minimum necessary lookahead depth is used. We provided statistics demonstrating that the number of requests for lookahead by our $SLL(k)$ determinism algorithm is a linear function of k. Although, each $LOOK_k$ request is exponentially complex, the number of grammar constructs that require $LOOK_k$ computations is small and, hence, our method of $SLL(k)$ analysis has a typical execution time that is practical as shown by Figure 6.10.

6.4 $SLL(k)$ Parser Construction

Constructing $SLL(k)$ parsers is a process of computing **induces** relations, constructing heterogeneous decision states, and building executable programs that implement the heterogeneous states. This section describes how information from **induces** relations can be reduce in size and represented as heterogeneous decision states. We present a number of different executable decision state implementations followed by two example parser constructions.

6.4.1 Lookahead Information Compression

Lookahead information for conventional $SLL(k)$ decisions has space complexity $O(|T|^k)$. This exponentiality can be reduced by using the minimum possible lookahead depth, k, and by employing $SLL^1(k)$ decisions whenever possible. In the event that $SLL^1(k)$ is insufficient, full $SLL(k)$ decisions must be constructed. However, heavy compression is possible even for these decisions.

Consider the generic $SLL(k)$ **induces** relation in Table 6.3 where m is the number of productions of A and k is the minimum necessary lookahead.

Table 6.3 Generic $SLL(k)$ **induces** Relation for Nonterminal A

Lookahead $(\tau_1, ..., \tau_k) \in T^k$	Action
$LOOK_k(A \rightarrow \bullet\, \alpha_1)$	*predict* $A \rightarrow \alpha_1$
$LOOK_k(A \rightarrow \bullet\, \alpha_2)$	*predict* $A \rightarrow \alpha_2$
...	...
$LOOK_k(A \rightarrow \bullet\, \alpha_m)$	*predict* $A \rightarrow \alpha_m$

Using an $SLL^1(k)$ decision offers the most significant reduction because $SLL^1(k)$ lookahead decisions have size $O(|T| \times k)$. The generic $SLL^1(k)$ **induces** relation is shown in Table 6.4.

Table 6.4 Generic $SLL^1(k)$ **induces** Relation for Nonterminal A

Lookahead $\tau_1, ..., \tau_k \in T, T, ..., T$	Action
$LOOK_1^1(A \rightarrow \bullet\, \alpha_1), ..., LOOK_k^1(A \rightarrow \bullet\, \alpha_1)$	*predict* $A \rightarrow \alpha_1$
$LOOK_1^1(A \rightarrow \bullet\, \alpha_2), ..., LOOK_k^1(A \rightarrow \bullet\, \alpha_2)$	*predict* $A \rightarrow \alpha_2$
...	...
$LOOK_1^1(A \rightarrow \bullet\, \alpha_m), ..., LOOK_k^1(A \rightarrow \bullet\, \alpha_m)$	*predict* $A \rightarrow \alpha_m$

When an $SLL^1(k)$ decision is not possible, $SLL(k)$ must be used. To compress full $SLL(k)$ information, we consider when $SLL^1(k)$ decisions are insufficient: $SLL^1(k)$ decisions are insufficient when an artificial lookahead k-string for a production, created by $SLL^1(k)$ compression, collides with a real tuple from another production's $LOOK_k$ set. A $SLL^1(k)$ decision can be augmented to test for the ''offending'' k-strings as a special case, thus, resolving the nondeterminism; this *hybrid decision* is an $SLL(k)$ decision because it uses k-tuples, but has much smaller space requirements than the conventional $SLL(k)$ decision in practice.

In general, for any production pair $A \rightarrow \alpha$ and $A \rightarrow \beta$, the number of real tuples from $LOOK_k(A \rightarrow \bullet\, \beta)$ that collide with artificial tuples resulting from $LOOK_i^1(A \rightarrow \bullet\, \alpha)$ for $1 \leq 1 \leq k$ is smaller than the full $LOOK_k(A \rightarrow \bullet\, \alpha)$ set. As the degenerate case, when $SLL(k)$ reduces to $SLL^1(k)$, the number of real tuples from $LOOK_k(A \rightarrow \bullet\, \beta)$ that collide with artificial tuples from $LOOK_i^1(A \rightarrow \bullet\, \alpha)$ for $1 \leq 1 \leq k$ is zero. On the other extreme, every real tuple from $LOOK_k(A \rightarrow \bullet\, \beta)$ could collide with artificial tuples from $LOOK_i^1(A \rightarrow \bullet\, \alpha)$ for $1 \leq 1 \leq k$, which renders the two productions purely $SLL(k)$ separable; a hybrid decision for these two productions is futile as only k-tuple comparisons are sufficiently powerful.

Computing a discriminant (separating) function for predicting productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ is done by computing the $\Delta_k(A, \alpha, \beta)$ and $\Delta_k(A, \beta, \alpha)$ discriminant k-tuple sets, which are the artificial tuples from $A \rightarrow \alpha$ that collide with real tuples of $A \rightarrow \beta$ and vice versa. Define

$$\Delta_k(A, \alpha, \beta) = LOOK_k(A \rightarrow \bullet \beta) \bigcap \text{artificial-tuples}(\Lambda^{\alpha})$$

with

$$\Lambda^{\alpha} = \{ LOOK_1^1(A \rightarrow \bullet \alpha), LOOK_2^1(A \rightarrow \bullet \alpha), ..., LOOK_k^1(A \rightarrow \bullet \alpha) \} \text{fP}$$

and

$$\text{artificial-tuples}(\Lambda^{\alpha}) = \text{tuples} (\Lambda^{\alpha}) - LOOK_k(A \rightarrow \bullet \alpha)$$

where the function ‘‘tuples’’ returns the set of k-tuples generated by permutations of the Λ_i sets and ‘‘artificial-tuples’’ is the set of tuples generated by the Λ_i sets, but which do not correspond to valid lookahead k-tuples.

Decision states for nonterminals with only two productions, $A \rightarrow \alpha$ and $A \rightarrow \beta$, are then easily constructed via the template in Figure 6.13.

upon $\tau_1 \in \Lambda_1^{\alpha}$ and $\tau_2 \in \Lambda_2^{\alpha}$ and ... and $\tau_k \in \Lambda_1^{\alpha}$ and $(\tau_1, \tau_2, \tau_3) \notin \Delta_k(A, \alpha, \beta)$ *predict* $A \rightarrow \alpha$;
upon $\tau_1 \in \Lambda_1^{\beta}$ and $\tau_2 \in \Lambda_2^{\beta}$ and ... and $\tau_k \in \Lambda_k^{\beta}$ and $(\tau_1, \tau_2, \tau_3) \notin \Delta_k(A, \beta, \alpha)$ *predict* $A \rightarrow \beta$;

Figure 6.13 Hybrid State for Nonterminal with Two Productions

When both Δ_k are $\varnothing$, the decision state reduces to an $SLL^1(k)$ state. When $\Delta_k(A, \alpha, \beta) = LOOK_k(A \rightarrow \bullet \beta)$ and $\Delta_k(A, \beta, \alpha) = LOOK_k(A \rightarrow \bullet \alpha)$, the decision is a purely $SLL(k)$ decision; it reduces to that in Figure 6.14 where the ‘‘$(\tau_1, \tau_2, \tau_3) \notin \Delta_k$’’ expressions have been replaced by the appropriate $LOOK_k$ sets.

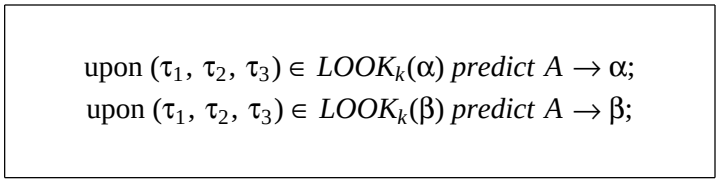


Figure 6.14 Purely *SLL* (*k*) State for Nonterminal with Two Productions

The use of Δ discriminant sets is advantageous in practice, but can perform unnecessary tuple comparisons in the worst case. Consider the maximum size of the discriminant tuple sets.

Lemma 6.1: *For any production* $A \rightarrow \alpha_i$, $| \bigcup_j \Delta_k(A, \alpha_i, \alpha_j) | < | \bigcup_j LOOK_k(\alpha_j) |$.

Proof:

By definition $\Delta_k(A, \alpha_i, \alpha_j) = LOOK_k(A \rightarrow \bullet \alpha_j) \bigcap$ artificial-tuples(Λ^{α}), which is clearly no larger than $LOOK_k(A \rightarrow \bullet \alpha_j)$. Hence, the combined size of all Δ sets is no greater than the combined size of all *LOOK* sets. Because $\Delta_k(A, \alpha_i, \alpha_j) = \varnothing$, the combined size of the Δ sets is strictly smaller than the combined size of all *LOOK* sets.

□

Lemma 6.1 indicates that, when there are exactly two alternative productions that are not $SLL^1(k)$ separable, the use of discriminant sets is always beneficial; the Δ sets can be no worse that doing tuple membership operations with *LOOK* sets. In the extreme, all *m* productions for a nonterminal are non-$SLL^1(k)$ separable; each prediction expression would be bounded by the combined size of all *LOOK* sets. Therefore, worst-case, the hybrid *SLL* (*k*) decision state with *m* alternative productions is *m* times as large as a normal *SLL* (*k*) (ignoring the relatively small cost of doing the $m \times k$ set comparisons). The break-even point occurs when the combined size of the required Δ sets equals the combined size of the *m LOOK* sets. In practice, Δ_k sets are much smaller than $LOOK_k$ sets because the lookahead overlap between alternative productions is typically low.

The relationship between $SLL^1(k)$ states and *SLL* (*k*) Δ_k sets is characterized by Theorem 6.1.

Theorem 6.1: if $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$ are $SLL^1(k)$ separable, then $\Delta_k(A, \alpha_i, \alpha_j) = \Delta_k(A, \alpha_j, \alpha_i) = \varnothing$ for some *i* and *j*.

Proof:
For two productions to be $SLL^1(k)$ separable, Λ_n^i and Λ_n^j must be disjoint for some lookahead depth $n \leq k$ by Lemma 6.1. This implies that there are no real or artificial tuples in common between the lookahead sets for both productions. Hence, there can be no real tuples derived from $A \rightarrow \alpha_j$ that collide with artificial tuples derived from $A \rightarrow \alpha_i \Rightarrow \Delta_k(A, \alpha_i, \alpha_j) = \varnothing$. Similarly, there cannot be a real tuple derived from $A \rightarrow \alpha_i$ that collides with an artificial tuple derived from $A \rightarrow \alpha_j \Rightarrow \Delta_k(A, \alpha_j, \alpha_i) = \varnothing$.
□

Theorem 6.1 provides sufficient conditions for an $SLL(k)$ state construction algorithm to avoid computation of Δ discriminant sets. In effect, the Theorem 6.1 and Lemma 6.1 suggest that $SLL^1(k)$ should be attempted before $SLL(k)$ and, further, when $SLL(k)$ is required, hybrid $SLL^1(k)/SLL(k)$ states should be constructed as they are typically smaller than equivalent normal $SLL(k)$ states; hence, $SLL^1(k)$ analysis is performed for every decision point in the grammar, never having been done unnecessarily.

Given a nonterminal, $SLL(k)$ heterogeneous state construction is accomplished via the algorithm in Figure 6.15.

procedure *constructSLL*(A : *nonterminal*, Λ : **array of sets**);
begin
    add ‘‘$\tau_1 \in \Lambda_1^{\alpha_i}$ and $\tau_2 \in \Lambda_2^{\alpha_i}$ and ... and $\tau_k \in \Lambda_1^{\alpha_i}$’’ to prediction expression for $A \rightarrow \alpha_i$;
    add ‘‘$\tau_1 \in \Lambda_1^{\alpha_j}$ and $\tau_2 \in \Lambda_2^{\alpha_j}$ and ... and $\tau_k \in \Lambda_1^{\alpha_j}$’’ to prediction expression for $A \rightarrow \alpha_j$;
    for each non-$SLL^1(k)$ production pair $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$ **begin**
        $f_1 = LOOK_k(A \rightarrow \bullet \alpha_i)$;
        $f_2 = LOOK_k(A \rightarrow \bullet \alpha_j)$;
        n = minimum lookahead depth such that productions i and j are $SLL(n)$ separable;
        d_1 = ‘‘$(\tau_1, \tau_2, \tau_3) \in \Delta_n(A, \alpha_i, \alpha_j)$’’;
        d_2 = ‘‘$(\tau_1, \tau_2, \tau_3) \in \Delta_n(A, \alpha_j, \alpha_i)$’’;
        add d_1 to prediction expression for $A \rightarrow \alpha_i$;
        add d_2 to prediction expression for $A \rightarrow \alpha_j$;
    end;
end *constructSLL*;

Figure 6.15 Algorithm on Grammar to Construct $SLL(k)$ Decision States

The construction algorithm assumes no order of testing when computing the prediction expressions; i.e. each expression is completely self-contained and does not require the results of previous tests. Although this can be a necessary feature for many implementation algorithms, a much smaller state can be constructed if the expressions are executed in a particular order. For example, if the expressions are guaranteed to execute first to last, then the expression for production $A \rightarrow \alpha_i$ has no need to examine the $\Delta_k(A, \alpha_i, \alpha_j)$ for $j = 1..i-1$. Further, they do not need to

be computed by the construction algorithm, which increases grammar analysis speed; even if computed needlessly, a decision state implementation mechanism can simply ignore the unneeded Δ sets.

6.4.2 Implementation of Heterogeneous Decision States

Lookahead decision implementations can be categorized as either *m*-ary or non-*m*-ary where an *m*-ary implementation maps a terminal lookahead sequence to one of *m* **induces** production predictions using a single prediction expression and are of the form $M[\tau_1,...,\tau_n]$ where $n \leq k$ and *M* maps the lookahead to a unique production. On the other extreme, a series of *m* tests (prediction expressions) can be made, one for each production, to see which production is predicted by the current lookahead buffer. Table 6.5 summarizes the popular *m*-ary decision implementation techniques.

Table 6.5 Implementation Strategies for *m*-ary Lookahead Decisions

Table[τ_1, ..., τ_k]	This is the obvious solution, but has huge space requirements, which incidentally, is what probably led others to consider $k>1$ lookahead impractical. Even sparse-matrix and table compression techniques can be large or impractically slow. This technique has time $O(k)$ and space $O(T^k)$ without compression.
Hash table	This technique holds much more promise than a straight table, but can also require large tables to obtain good performance. An interesting direction in this area is perfect hashing techniques which might be applied to good effect. This technique has time $O(k)$ and the space requirements can reach $O(T^k)$ in the worst-case.
Token renumbering	This method temporarily remaps terminals to different values to obtain set of hyperplanes; simple, non-overlapping regions are created in tuple space. Computing a terminal translation mapping could be expensive, but a simple relational operator could be used to predict productions. This technique has time $O(k)$ and space $O(\|T\|^k)$ at parser run-time (if a terminal remapping exists).

These techniques are not feasible for *SLL* (*k*) due to exponential lookahead space requirements. If a series of tests is made, the size of a lookahead decision state can be dramatically reduced at the cost of a small reduction in parser speed. Table 6.6 summarizes a few of the options in non-*m*-ary decision implementations.

Table 6.6 Implementation Strategies for Non-*m*-ary Lookahead Decisions

Decision tree	Using a (nearly balanced) decision tree can be used to make an $\log m$ mapping. To use this method, an ordering must be established to allow traversal of the tree. This method has time complexity $O(\log m)$ and space complexity of $O(\log m \times \lvert T \rvert^k)$. Note that a tree would roughly the same as a lookahead DFA.
Series of Tests	As with the decision tree, a series of tests can be much cheaper than a single decision. On average, this implementation is $O(m)$ because most decisions are $SLL(1)$ where m is the number of productions for a nonterminal. This decision mechanism is trivial to construct. No ordering is necessary for this method to find the correct production. Tests can be simpler than the tree method because more tests are performed on average. This method has the additional benefit that the productions with highest frequency of application can be specified first to decrease average prediction time. Also, each test in the sequence may use a different lookahead depth. Time is $O(m \times \lvert T \rvert^k)$ in the worst case, but the hybrid $SLL^1(k)/SLL(k)$ mechanism can be used as described.

For $SLL(k)$, m-ary tests are infeasible because the hybrid $SLL^1(k)/SLL(k)$ states, which appear to be the best way to build practical $SLL(k)$ parsers, are not simple k-tuple to action mappings. On the other hand, one could build small hash tables to test for Δ set membership. In our experience, gained mainly from PCCTS, full $SLL(k)$ decisions are rare and of those full $SLL(k)$ decisions, the Δ sets are few and small — hash tables are overkill. Therefore, as with $SLL^1(k)$ we advocate a series of tests rather than one of the m-ary decision mechanisms and employ recursive-descent parsers because of their flexibility.

6.4.3 Example $SLL(k)$ Parser Constructions

We introduce recursive-descent parser construction via grammar Grammar G6.3.

$$\begin{aligned} A &\rightarrow B \\ A &\rightarrow ad \\ B &\rightarrow ab \\ B &\rightarrow cd \end{aligned} \qquad \text{G6.3}$$

The $SLL^1(2)$ induces relation is given in Figure 6.7.

Table 6.7 $SLL^1(2)$ **induces** Relation for Nonterminal A in Grammar G6.3

Lookahead $\tau_1,\tau_2 \in T,T$	Action
$\{a,c\},\{b,d\}$	*predict* $A \rightarrow B$
$\{a\},\{d\}$	*predict* $A \rightarrow ad$

The two productions of A are not $SLL^1(2)$ separable because no single lookahead depth can be used to distinguish between the productions. A parser-generator must then turn to $SLL(2)$. The appropriate hybrid state, as computed by *constructSLL*, is given in Figure 6.16.

upon $\tau_1 \in \{a,c\}$ and $\tau_2 \in \{b,d\}$ and $(\tau_1, \tau_2) \neq (a,d)$ *predict* $A \rightarrow B$;
upon $\tau_1 = a$ and $\tau_2 = d$ *predict* $A \rightarrow ad$;

Figure 6.16 Hybrid *SLL* (*k*) State for Nonterminal *A* of Grammar G6.3

The $\Delta_2(A, ad, B) = \varnothing$ is not included for the second prediction expression as there are no artificial tuples generated by $\Lambda^{ad} = \{a\},\{b\}$. A recursive-descent procedure that would implement nonterminal *A* is given in Figure 6.17.

```
procedure A;
begin
    if τ1 ∈ {a,c} and τ2 ∈ {b,d} and (τ1,τ2) ≠ (a,d) then begin
        B;
    end;
    elseif τ1 = a and τ2 = d then begin
        MATCH(a);
        MATCH(d);
    end;
end A;
```

Figure 6.17 *SLL* (2) Implementation of *A* for Grammar 6.3

As mentioned above, the order of prediction expression evaluation can reduce the size of a parser. Figure 6.18 shows a functionally equivalent, but smaller parser.

```
procedure A;
begin
    if τ1 = a and τ2 = d then begin
        MATCH(a);
        MATCH(d);
    end;
    elseif τ1 ∈ {a,c} and τ2 ∈ {b,d} then begin
        B;
    end;
end A;
```

Figure 6.18 Alternate *SLL* (2) Implementation of *A* for Grammar 6.3

Because the special case of tuple (a,d) has already been tested for in the first prediction expression, the second does not need to consider it. To reiterate, the i^{th} prediction expression need not consider Δ sets associated with productions $1..i-1$.

As a more complicated example, consider Grammar G6.4.

$$\begin{aligned} A &\rightarrow B \\ A &\rightarrow C \\ B &\rightarrow ab \\ B &\rightarrow cd \\ C &\rightarrow ad \\ C &\rightarrow yb \\ C &\rightarrow cx \end{aligned} \qquad \text{G6.4}$$

Table 6.8 *SLL* (2) **induces** Relation for Nonterminal *A* in Grammar G6.4

Lookahead $(\tau_1,\tau_2) \in T^2$	Action
(a,b)	*predict* $A \rightarrow B$
(c,d)	*predict* $A \rightarrow B$
(a,d)	*predict* $A \rightarrow C$
(y,b)	*predict* $A \rightarrow C$
(c,x)	*predict* $A \rightarrow C$

Table 6.9 $SLL^1(2)$ **induces** Relation for Nonterminal A in Grammar G6.4

Lookahead $\tau_1, \tau_2 \in T, T$	Action
$\{a,c\},\{b,d\}$	*predict* $A \rightarrow B$
$\{a,y,c\},\{b,d,x\}$	*predict* $A \rightarrow C$

The set of artificial tuples for $A \rightarrow B$ is $\{(a,d),(c,b)\}$ and $\{(a,b),(a,x),(y,d),(y,x),(c,d),(c,b)\}$ for $A \rightarrow C$. There are three collisions to worry about: A Real tuple for $A \rightarrow C$, (a,d), collides with an artificial tuple for $A \rightarrow B$, both valid tuples for $A \rightarrow B$ collide with artificial tuples of $A \rightarrow C$, and both productions have (c,b) as an artificial tuple.

upon $\tau_1 \in \{a,c\}$ and $\tau_2 \in \{b,d\}$ and $(\tau_1, \tau_2) \neq (a,d)$ *predict* $A \rightarrow B$;
upon $\tau_1 \in \{a,y,c\}$ and $\tau_2 \in \{b,d,x\}$ and $(\tau_1, \tau_2) \notin \{(a,b),(c,d)\})$ *predict* $A \rightarrow C$;

Figure 6.19 Hybrid State for Nonterminal A in Grammar G6.4

A recursive-descent implementation procedure for nonterminal A is provided in Figure 6.20.

```
procedure A;
begin
    if τ1 ∈ {a,c} and τ2 ∈ {b,d} and (τ1, τ2) ≠ (a,d) then begin
        B;
    end;
    elseif τ1 ∈ {a,y,c} and τ2 ∈ {b,d,x} then begin
        C;
    end;
end A;
```

Figure 6.20 Hybrid $SLL^1(2)$/$SLL(2)$ Implementation of A for Grammar 6.4

Notice that the special case test '' $(\tau_1, \tau_2) \notin \{(a,b),(c,d)\}$'' is unnecessary due to the prediction-expression order of execution.

This chapter described how $SLL(k)$ parsers may be constructed. The main principles behind $SLL(k)$ parsing have long been understood from a theoretical point of view, but little practical work has been done because $SLL(k)$ parsing was considered intractable. We have demonstrated the practicality of $SLL(k)$ lookahead computation, testing grammars for the $SLL(k)$ property, and constructing $SLL(k)$ parsers for $k > 1$.

Section 6.16 provided three algorithms for computing $SLL(k)$ lookahead information that implement the recurrences in Section 4.9.1. The lookahead for a particular position, *p*, in a grammar is computed by walking the associated GLA collecting the non-ε edges along paths emanating from the GLA state created for position *p*. The straightforward algorithm had two exponential terms in its complexity: $|G|^k$ from the recursive nature of grammars, which forces redundant computations, and $|T|^k$ from the worst-case size of lookahead information. The constrained algorithm reduced the typical size of the lookahead information by constraining its walk of the GLA to only those paths that can possibly lead to *k*-strings in common between productions. The caching algorithm removed the grammar-derived exponentiality by saving the results of computations. The straightforward and constrained lookahead computation algorithms had $O(|G|^k \times |T|^k)$ time and space complexity in the worst case whereas the caching algorithm had $O(|G| \times k \times |T|^k)$.

Section 6.17 described how grammars may be tested for the $SLL(k)$ property by comparing the lookahead information of each production. By minimizing the lookahead depth, *k*, and by first testing for the $SLL^1(k)$ property, the time and space necessary to test grammars for the $SLL(k)$ property can be reduced significantly. In the worst case, our approach is dominated by the time required to compute the lookahead information and can be implemented in time and space $O(|G| \times k \times |T|^k)$.

Section 6.18 detailed the construction of practical $SLL(k)$ parsers for $k > 1$. We relied on heavy lookahead-information compression to avoid the worst-case space requirement, $O(|T|^k)$, of the lookahead information. Just as in $SLL(k)$ property testing, the minimum lookahead is used and $SLL^1(k)$ decisions are used before resorting to $SLL(k)$. Our compression techniques apply equally well to all $C(k)$ parsing strategies.

CHAPTER 7 *LALL* (*k*), *LL* (*k*), *SLR* (*k*), *LALR* (*k*), AND *LR* (*k*)

The previous chapters introduced new ways to represent lookahead information and grammars, provided means of computing strong lookahead information, presented algorithms that test grammars for the $SLL^1(k)$ and $SLL(k)$ property, and described methods for constructing $SLL^1(k)$ and $SLL(k)$ parsers. The strong class of $LL(k)$ parsers was emphasized because the lookahead computation and parser construction mechanisms are the simplest, yet still effectively demonstrate the important issues in the proposed construction methods. In this chapter, we explore the other variants of $LL(k)$ and outline how $LR(k)$ and its variants can take advantage of the techniques emphasized in this thesis. Further, we generalize these parsers to $LL^m(k)$ and $LR^m(k)$.

We begin by finishing off the $LL(k)$-based classes. The first section describes $LALL(k)$ [SiS82], which is perhaps the most useful of the $LL(k)$ classes. With little modification, the $SLL(k)$ algorithms can be applied to $LALL(k)$. The second section describes full $LL(k)$. The linear approximation to $LL(k)$, $LL^1(k)$, is of little use during grammar analysis, but proves very useful during decision state construction as a compression technique. The remainder of the chapter discusses the $LR(k)$ variants beginning with $SLR(k)$. The fourth section demonstrates that the $LOOK_k$ and $LOOK_k^1$ computations given for $SLL(k)$ may be used directly for $SLR(k)$. The fifth section shows how $LALR(k)$, like $LALL(k)$, can use the techniques of linear lookahead approximation to reduce decision state size. The sixth section describes full $LR(k)$ and how, like full $LL(k)$, $LR^1(k)$ analysis is not useful except for decision state compression. Finally, we present the generalized parsers of $LL^m(k) \subseteq LL(k)$ and $LR^m(k) \subseteq LR(k)$, which use lookahead decisions whose largest unit of comparison is an m-tuple for $m \leq k$.

7.1 *LALL* (*k*)

$LALL(k)$ lies properly between the $SLL(k)$ and $LL(k)$ class of grammars [SiS82]; ANTLR employs such parsers [PDC92]. $LALL(k)$ is analogous to the $LALR(k)$ class of grammars — the parser resulting from the merging of all states of common core. Just as $LALR(k)$ parsers have the same number of states as $SLR(k)$ parsers derived from the same grammars, $LALL(k)$ parsers have the same number of states as $SLL(k)$ parsers. The difference lies in the accuracy of the lookahead information. As a result, testing for the $LALL(k)$ property is identical to testing for the $SLL(k)$ property except that the $LOOK_k$ and $LOOK_k^1$ computations will return $LALL(k)$ lookahead information, which is a subset of the $SLL(k)$ lookahead information. Parsers are constructed in

exactly the same manner as *SLL* (*k*) parsers, but again, using the more accurate *LALL* (*k*) lookahead information. Hence, this section merely provides the modifications to the *SLL* (*k*) *LOOK* algorithms necessary to compute *LALL* (*k*) information. We begin by describing the difference between *SLL* (*k*) and *LALL* (*k*) lookahead.

When *SLL* (*k*) analysis reaches the exit state of some nonterminal, *A*, in a GLA, it proceeds to compute the *LOOK* of all arcs emanating from the exit state. These *FOLLOW*-links point to the states following each reference to *A* in other nonterminals' states. Hence, *SLL* (*k*) analysis combines the results of all *FOLLOW* operations when, in reality, at most one nonterminal can reference *A* at a time; *SLL* (*k*) is a covering approximation to the real lookahead information. *SLL* (*k*) analysis is said to use context-insensitive *FOLLOW* information whereas *LALL* (*k*) and *LL* (*k*) analysis is said to use context-sensitive *FOLLOW* information. Two things can be done to improve the accuracy of *SLL* (*k*) lookahead. First, when computing $LOOK_n(A \rightarrow \alpha \bullet B\beta)$, $LOOK_{n'}(A \rightarrow \alpha B \bullet \beta)$ can be used rather than $FOLLOW(B)$ if *B* does not always generate *n*-strings where $n' \leq n$. Second, prediction expressions of a nonterminal's productions can be made dependent on the context of the reference to that nonterminal; alternatively, the normal *LL* (*k*) to *SLL* (*k*) conversion can be used, which makes nonterminal references unique by duplication. *LALL* (*k*) embodies the first improvement and *LL* (*k*) incorporates both.

LALL (*k*) lookahead is more easily seen by example. Consider Grammar G7.1, which is *LALL* (2), but not *SLL* (2).

$$\begin{aligned} A &\rightarrow aBc \\ A &\rightarrow ad \\ A &\rightarrow C \\ B &\rightarrow b \\ B &\rightarrow \\ C &\rightarrow cBd \end{aligned} \qquad \text{G7.1}$$

The partial *SLL* (2) **induces** relation is shown in Table 7.1.

Table 7.1 *SLL* (2) **induces** Relation for Nonterminal *A* in Grammar G7.1

Lookahead $(\tau_1, \tau_2) \in T^2$	Action
(a,b)	*predict* $A \rightarrow aBc$
(a,c)	*predict* $A \rightarrow aBc$
(a,d)	*predict* $A \rightarrow aBc$
(a,d)	*predict* $A \rightarrow ad$
(c,b)	*predict* $A \rightarrow C$
(c,c)	*predict* $A \rightarrow C$
(c,d)	*predict* $A \rightarrow C$

Because (a,d) predicts both productions one and two of *A*, the *SLL* (2) **induces** is inconsistent. However, the sequence *ad* can never be generated by $A \rightarrow aBc$; the *SLL* (k) lookahead information is a covering superset of the actual lookahead. Tuple (a,d) arises from the fact that the *SLL* (2) analysis combined the symbols following all references to *B*. In fact, when called from *A*, *B* can only be followed by terminal *c*. The *LALL* (2) (and *LL* (2)) **induces** relation has the correct predictions; it is shown in Table 7.2.

Table 7.2 *LALL* (2) **induces** Relation for Nonterminal *A* in Grammar G7.1

Lookahead $(\tau_1, \tau_2) \in T^2$	Action
(a,b)	*predict* $A \rightarrow aBc$
(a,c)	*predict* $A \rightarrow aBc$
(a,d)	*predict* $A \rightarrow ad$
(c,b)	*predict* $A \rightarrow C$
(c,d)	*predict* $A \rightarrow C$

LALL (2) lookahead information is smaller and more accurate than *SLL* (2) lookahead, but comes at the cost of more complicated $LOOK_k$ and $LOOK_k^1$ algorithms.

To compute the more accurate *LALL* (k) lookahead sets, two modifications to the *LOOK* algorithms can be used, both of which, reduce the utility of the cache. The first is simpler, but makes caching exceedingly complicated and, thus, less attractive. It involves enabling and disabling the *FOLLOW*-links emanating from the nonterminal exit states. Unfortunately, the exit state caches would have to contain entries for each context in which the associated nonterminal could be referenced — an impractically large cache. The second method is a little more difficult,

but maintains a relatively straightforward caching mechanism. This modification requires a new lookahead tree node type and has caches only in the nonterminal entry states.

We describe the second $LALL(k)$ method by altering the uncached $SLL(k)$ $LOOK_k$ algorithm. Before $LOOK_k$ attempts to traverse an ε-arc in the GLA, resulting from a nonterminal reference, the exit node for that referenced nonterminal is marked as busy (the state of the busy flags is, naturally, saved before being marked). In this way, the $LOOK_k$ algorithm will not traverse any *FOLLOW*-links for nonterminal references; $LOOK_k$ will only compute the *FIRST* in this case. If $LOOK_k$ reaches the busy exit state, a special ε_n node is deposited in the tree in place of the actual *FOLLOW* subtree. This ε_n node is to be distinguished from the ε edges in the GLA; it is a place holder that indicates that $LOOK_n$ must be initiated on the GLA state following the nonterminal reference state. In this way, the context-sensitive *FOLLOW* is computed. There may be many ε_n nodes in the lookahead tree returned by a $LOOK_k$ invocation and the algorithm must replace each instance with the appropriate subtree. Upon returning from the ε-arc computation, $LOOK_k(p.edge_1)$ computes $LOOK_n(p.follow)$ for each ε_n node. Edge $p.follow$ is the node where parsing would continue after recognizing the nonterminal pointed to by $p.edge_1$; i.e. for $A \rightarrow \alpha \bullet_1 B \bullet_2 \beta$, $p.follow$ is the edge from p, the node created for position $\bullet_1$, to the node created for position $\bullet_2$. The modified algorithm is presented in Figure 7.1.

```
function LOOK_k( p : Node ) returns tree of terminal;
begin
    var t,u : tree of terminal;
              b : boolean;
              e : Node;

    if p==nil or k==0 then return nil;
    if p.busy[k] then return nil;
    p.busy[k] = true;
    if ( p.edge_1 is-a-terminal )
                  p.label_1
        t =           ↓          ; /* make label root of what follows */
            LOOK_{k-1}(p.edge_1)
    else begin
        e = exit-state-of-nonterminal-referenced(p.edge_1);
        b = e.busy[k];
        e.busy[k] = true;
        t = LOOK_k(p.edge_1);
        e.busy[k] = b;
        for each ε_n node, q, in t do
            replace-node-with-tree(q, LOOK_n(p.follow));
    end
    u = LOOK_k(p.edge_2);
    p.busy[k] = false;
    if t==nil then return u;
    else return t → u; /* t,u are siblings in tree */
end LOOK_k;
```

Figure 7.1 *LALL* (*k*) $LOOK_k$ Algorithm on GLA

The $LOOK_n(p.follow)$ computation may itself be barred from computing the *FOLLOW* if the current nonterminal itself was referenced by another. In general, only the $LOOK_k$ computation at the root of the recursive computation tree will be allowed to enter the exit state for the associated nonterminal, thus, computing a global *FOLLOW* only in this case. $LL(k)$ parsers solve this problem by splitting states. For example, $LOOK_k(B \rightarrow \bullet\, b)$ from Grammar G7.1, when not invoked from another *LOOK*, enters the exit state for B and effectively computes $FOLLOW(B) = \{c,d\}$. $LL(k)$ analysis would treat c and d separately as they occur in different contexts.

Caching the results of the $LALL(k)$ $LOOK_k$ is done by saving the results of $LOOK_k(p)$ in $p.cache[k]$ where p is the entry GLA state of some nonterminal; if that particular $LOOK_k$ is at the root of the computation tree, the results are not cached as the lookahead tree will contain *FOLLOW* information. Therefore, only the *FIRST* of a nonterminal is cached (ε_n nodes included). The $LOOK_k$ information does not contain *FOLLOW* information as it is a function of context; for any nonterminal the number of situations in which it can be referenced is exponentially large. Caching $LOOK_k$ for exit states is prohibitively expensive and is not done. Surprisingly, this limited caching mechanism does not render the $LALL(k)$ algorithm impractical; e.g., ANTLR, the parser generator of PCCTS, does not cache $LOOK_k$ information at all as it employs a constrained $LOOK_k$ algorithm similar to that of Section 6.16.3.

Although lookahead trees are smaller during $LALL(k)$ analysis, the lack of *FOLLOW* caching probably leaves $SLL(k)$ analysis faster, albeit less accurate. The constrained *LOOK* algorithms cannot cache results as they compute lookahead information that is restricted to a subset of the real lookahead; however, since the constrained approach avoids a large number of the *LOOK* computations, it proves practical. Formally, $LALL(k)$ parsers, resulting from this improved analysis, are superior to $SLL(k)$ parsers.

Theorem 7.2: $SLL(k) \subset LALL(k) \subset LL(k)$ *for* $k > 1$ [SiS90].

In summary, $LALL(k)$ analysis computes more accurate lookahead than $SLL(k)$ analysis by using context-sensitive versus context-insensitive *FOLLOW* information during *LOOK* computations for nonterminal references. $LALL(k)$ parsers are typically smaller than $SLL(k)$ parsers and have greater recognition strength; all of the $SLL(k)$ parser construction mechanisms are immediately applicable to $LALL(k)$ parser construction. The associated analysis algorithms are only slightly more complicated and are, perhaps, a bit slower; the linear approximation methods, $LOOK_k^1$, may still be applied to reduce analysis time. We conclude that $LALL(k)$, which is nearly $LL(k)$, should be employed for parser generators designed for widespread use. We observe, finally, that the one situation in which $LALL(k)$ analysis is identical to $SLL(k)$ analysis is the special case which renders $LALL(k)$ weaker than $LL(k)$.

7.2 *LL* (*k*)

The *LALL* (*k*) parsers, described in the previous section, differ from *SLL* (*k*) parsers only in the lookahead information; i.e. *LALL* (*k*) analysis yields more accurate lookahead sets. As a result, methods used to test grammars for *LALL* (*k*) property and methods for constructing *LALL* (*k*) parsers are identical to the *SLL* (*k*) techniques. *LL* (*k*) parsers, on the other hand, are exponentially large, have more complicated analysis and construction algorithms, and are typically unable to use the linear approximation methods during analysis. *LL* (*k*) parsers are impractical for these reasons, but nonetheless we describe, for completeness, how *LL* (*k*) differs from *LALL* (*k*) and how the linear approximation techniques described throughout this thesis can be applied to reduce lookahead decision state size.

In general, *LALL* (*k*) lookahead computations only yield lookahead sequences that can be generated by application of the production to be predicted. However, when predicting a production of some nonterminal *A* that can generate strings shorter than the required lookahead, the symbols following references to *A* must be used in the prediction. Determining which symbols to include can only be determined at parser run-time because it depends on the context in which *A* was invoked. During analysis only the set of possibilities is known. If there are *r* references to *A*, then at least *r* distinct lookahead sets must be available to predict the production; in general, the number of contexts is an exponential function of the grammar size. The parser state (context) will determine which of the lookahead sets to use when *A* is invoked. *LALL* (*k*) (and *SLL* (*k*)) analysis merges all of the possible following terminal sequences and, hence, do not use context to predict productions at parser run-time; this is the source of *LALL* (*k*)'s inferiority to *LL* (*k*).

The improved analysis of *LL* (*k*) can be incorporated into parsers by having multiple copies of the states that predict the productions of *A* — one copy for each context. Another method, which is effective for recursive-descent implementations, requires each procedure to define a parameter that represents the context in which the procedure was invoked. A third method, which operates on the grammar, makes each nonterminal reference unique by duplication and then applies *SLL* (*k*) analysis; see [FiL88]. For our purposes, we choose a recursive-descent implementation to demonstrate this improved lookahead scheme.

Consider Grammar G7.2 which is *LL* (2), but not *LALL* (2).

$$
\begin{aligned}
&A \rightarrow xBa \\
&A \rightarrow C \\
&B \rightarrow b \\
&B \rightarrow \\
&C \rightarrow yBba
\end{aligned}
\qquad \text{G7.2}
$$

The *LALL* (2) **induces** relation nonterminal *B* of this grammar is given in Table 7.3.

Table 7.3 *LALL* (2) **induces** Relation for Nonterminal *B* in Grammar G7.2

Lookahead $(\tau_1, \tau_2) \in T^2$	Action
(b,b)	*predict* $B \rightarrow b$
(b,a)	*predict* $B \rightarrow b$
(b,a)	*predict* $B \rightarrow$
$(a,\$)$	*predict* $B \rightarrow$

This is obviously non-*LALL* (2) as (b,a) predicts both productions of *B*. However, when *B* is invoked from *A*, *ba* can only be recognized by applying the first production; when *B* is invoked from *C*, *ba* can only be recognized by applying the second production. Context removes the *LALL* (2) inconsistency, hence, the decision is *LL* (2). Table 7.4 provides the *LL* (2) **induces** relation including context information.

Table 7.4 *LL* (2) **induces** Relation for Nonterminal *B* in Grammar G7.2

Context	Lookahead $(\tau_1, \tau_2) \in T^2$	Action
2	(b,b)	*predict* $B \rightarrow b$
1	(b,a)	*predict* $B \rightarrow b$
2	(b,a)	*predict* $B \rightarrow$
1	$(a,\$)$	*predict* $B \rightarrow$

Using the full *LL* (2) information, Figure 7.2 provides a procedure that correctly implements *B*.

```
procedure B(ctxt:integer);
begin
    if (ctxt=1 and τ1 = b and τ2 = a) or (ctxt=2 and τ1 = b and τ2 = b) then begin
        MATCH(b);
    end;
    elseif (ctxt=1 and τ1 = a and τ2 = $) or (ctxt=2 and τ1 = b and τ2 = a) then begin
    end;
end B;
```

Figure 7.2 *LL* (2) Implementation of *B* in Grammar 7.2

Each reference to *B* is given a unique integer that represents context; here, context is deceptively simple as only two values are required. In general, context information for *B* would encode the entire path from start symbol to the invocation of *B*. There are, unfortunately, an exponential number of possible contexts; this is the source of *LL* (*k*) impracticality.

Although they are generally impractical, *LL* (*k*) parsers can still take advantage of the linear approximation techniques. Consider an *LL* (*k*) decision state with *m* possible transitions. Let t_i represent the *k*-deep lookahead tree that induces the i^{th} transition. The compressed lookahead sets are trivially computed by merging all lookahead terminals at each depth *j* into Λ_j^i for $1 \le j \le k$. As before, if there is a lookahead depth that separates each transition-pair, the decision is $LL^1(k)$; in this case, the decision state would be linear in *k* rather than exponential. When it is not $LL^1(k)$, we employ the Δ_k discriminant sets exactly as used for *SLL* (*k*); inside a decision state there is no difference between *SLL* (*k*), *LALL* (*k*) and *LL* (*k*) (or any of the *LR* (*k*) variants).

LL (*k*) parsers are stronger than *SLL* (*k*) and *LALL* (*k*) parsers, but are exponential in size; the lookahead decision states may be reduced to near linear size, but the number of states will always be exponential in the worst case. We have shown how *LL* (*k*) lookahead differs from *LALL* (*k*) lookahead and described how this lookahead can be compressed using $LL^1(k)$ and hybrid $LL^1(k)$/*LL* (*k*) decisions as done for *SLL* (*k*) decision states.

7.3 *SLR* (*k*)

Although *SLR* (*k*) parsers are very different from *SLL* (*k*) parsers, the lookahead computations are identical and the *LOOK* algorithms may be used without modification; *SLR* (*k*) analysis uses ‘‘global’’ *FOLLOW* information just like *SLL* (*k*) analysis. As a result, the linear approximation $LOOK_k^1$ algorithm can be used to reduce *SLR* (*k*) analysis time. As an example, consider Grammar 7.3 which is *SLR* (2); we will also use this grammar during the discussions of *LALR* (*k*) and *LR* (*k*).

$$
\begin{aligned}
&S \rightarrow AB\$\$ \\
&A \rightarrow C \\
&A \rightarrow \\
&B \rightarrow a \\
&B \rightarrow b \\
&C \rightarrow aAx
\end{aligned}
\qquad \text{G7.3}
$$

A portion of the *SLR* (2) machine is shown in Figure 7.3.

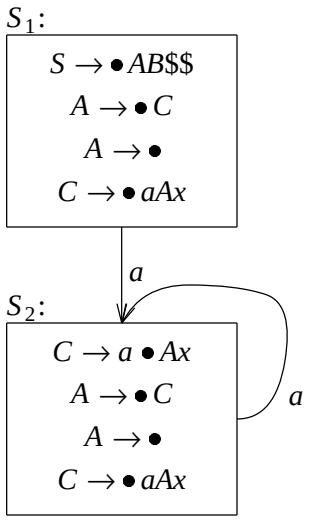


Figure 7.3 Partial *SLR* (2) Machine for Grammar G7.3

SLR (*k*) parser construction algorithm is different from *SLL* (*k*), but, once again, the linear approximation mechanism can be applied to reduce the size of *SLR* (*k*) decision states; this property will hold true for *LALR* (*k*) and *LR* (*k*) as well. Table 7.5 describes the normal *SLR* (2) action table for the partial *SLR* (2) machine as encoded by an **induces** relation.

Table 7.5 *SLR* (2) **induces** Relation for Partial *SLR* (2) Machine For Grammar G7.3

Action Table		
State	Lookahead $(\tau_1, \tau_2) \in T^2$	Action
S_1	$(a, \$)$	*reduce* $A \rightarrow$
	$(b, \$)$	*reduce* $A \rightarrow$
	(x,a)	*reduce* $A \rightarrow$
	(x,b)	*reduce* $A \rightarrow$
	(x,x)	*reduce* $A \rightarrow$
	(a,a)	*shift, goto* S_2
	(a,b)	*shift, goto* S_2
	(a,x)	*shift, goto* S_2
S_2	$(a, \$)$	*reduce* $A \rightarrow$
	$(b, \$)$	*reduce* $A \rightarrow$
	(x,a)	*reduce* $A \rightarrow$
	(x,b)	*reduce* $A \rightarrow$
	(a,a)	*shift, goto* S_2
	(a,b)	*shift, goto* S_2
	(a,x)	*shift, goto* S_2

The ‘‘goto’’ table, which maps nonterminals to states, is not included because it does not depend on lookahead information. The $SLR(2)$ action table, however, maps a lookahead sequence to a *shift* or a *reduce* action. $SLR(k)$ lookahead computations may use the $SLL(k)$ $LOOK_k$ directly. Notice that $LOOK_k(A \rightarrow \bullet)$, as defined in Chapter 6, is exactly the set of lookahead sequences that induces a *reduce* $A \rightarrow$ action and that $LOOK_k(C \rightarrow \bullet aAx)$ is exactly the lookahead set that induces a *shift, goto* S_2 action.

The linear approximation $LOOK_k^1$ algorithm can also be applied for $SLR(k)$ grammars without modification to reduce analysis time. Grammar G7.3 is $SLR(2)$ and $SLR^1(2)$. Consider the $SLR^1(2)$ **induces** relation in Table 7.6, which could be computed via $LOOK_k^1$ or by simply computing Λ_i sets (compressing all terminals at lookahead depth i for all i).

Table 7.6 $SLR^1(2)$ **induces** Relation for Partial $SLR(2)$ Machine For Grammar G7.3

Action Table		
State	Lookahead $\tau_1, \tau_2 \in T, T$	Action
S_1	$\{a,b,x\}$, $\{a,b,x,\$\}$ $\{a\}$, $\{a,b,x\}$	*reduce* $A \rightarrow$ *shift, goto* S_2
S_2	$\{a,b,x\}$, $\{a,b,x,\$\}$ $\{a\}$, $\{a,b,x\}$	*reduce* $A \rightarrow$ *shift, goto* S_2

The $SLR^1(2)$ compressed **induces** relation is not deterministic as there is no lookahead depth that is disjoint for the lookahead/action pair (interestingly, the $LALR^1(2)$ and $LR^1(2)$ **induces** relations are deterministic). As with the $LL(k)$ variants, the hybrid state techniques of Chapter 6 can be used to reduce the size of the lookahead information even though they are not $SLR^1(k)$. For example, state S_1 can be implemented as the heterogeneous automaton state in Figure 7.4.

upon $\tau_1 \in \{a,b,x\}$ and $\tau_2 \in \{a,b,x,\$\}$ and $(\tau_1, \tau_2) \notin \{(a,a),(a,b),(a,x)\}$ *reduce* $A \rightarrow$;
upon $\tau_1 = a$ and $\tau_2 \in \{a,b,x\}$ *shift, goto* S_2;

Figure 7.4 Heterogeneous Automaton State for State S_1 of Figure 7.3

To store lookahead information, the state in Figure 7.4 requires three terminal sets (at $|T|/wordsize$ words each), 1 terminal, and three 2-tuples for a total of $O(3 \times |T| / wordsize + 1 + 6)$ (about 7) words. Without the hybrid technique, a normal state would need to store eight 2-tuples for a total of 16 words. The $O(|T|^k)$ storage requirements for a conventional parser state quickly surpass the ‘‘sets plus a few tuples’’ approach of the $SLR^1(2)$/$SLR(2)$ state.

$SLR(k)$ is not commonly used, but its relationship to $SLL(k)$, with regards to lookahead and decision states, is interesting — $SLL(k)$ lookahead computation algorithms may be used directly. Accordingly, since $SLR(k)$ parsers have sizes linear in the grammar size, we observe that linear $SLR^1(k)$ parsers and near-linear $SLR(k)$ parsers can be obtained contrary to the conventional wisdom that they are always exponential due to lookahead information size.

7.4 $LALR(k)$

$LALR(1)$ parsers are very common due to the proliferation of $LALR(1)$ parser generators. $LALR(k)$, in contrast, is almost unknown for practical systems. As with $SLR(k)$, $LALR(k)$ parsers are considered an exponential problem due to the size of the lookahead information. However, the fact that $LALR(k)$ and $LALL(k)$ parsers are duals [SiS90] of each other, implies that $LALR(1)$ $LOOK_k^1$ lookahead computations may be defined and used to reduce grammar analysis time and lookahead decision state size.

Reconsider Grammar G7.3; Figure 7.5 shows the same portion of the $LALR(2)$ machine that Figure 7.3 shows of the $SLR(2)$ machine.

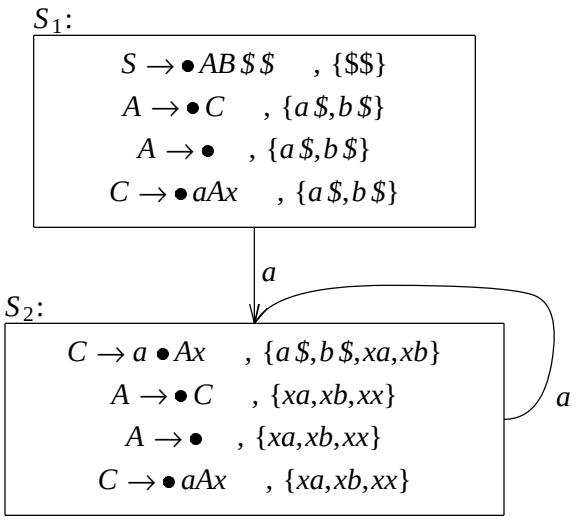


Figure 7.5 Partial *LALR*(2) Machine for Grammar G7.3

Table 7.7 describes the normal *LALR*(2) action table for the partial *LALR*(2) machine as encoded by an **induces** relation.

Table 7.7 *LALR*(2) **induces** Relation for Partial *LALR*(2) Machine For Grammar G7.3

Action Table		
State	Lookahead $(\tau_1, \tau_2) \in T^2$	Action
S_1	$(a, \$)$	*reduce* $A \rightarrow$
	$(b, \$)$	*reduce* $A \rightarrow$
	(a,a)	*shift, goto* S_2
	(a,b)	*shift, goto* S_2
S_2	(x,a)	*reduce* $A \rightarrow$
	(x,b)	*reduce* $A \rightarrow$
	(x,x)	*reduce* $A \rightarrow$
	(a,a)	*shift, goto* S_2
	(a,x)	*shift, goto* S_2

The lookahead information in Table 7.7 is a subset of the lookahead in Table 7.5; *SLR*(*k*) and *LALR*(*k*) parsers have the same states, but *LALR*(*k*) parsers have more accurate lookahead information (at the cost of more complicated grammar analysis algorithms). Grammar G7.3 is not $SLR^1(2)$, but it is $LALR^1(2)$. Again, we compress the lookahead via computation with $LALR^1(k)$

$LOOK_k^1$ or by computing Λ_i from the $LOOK_k$ information; see Table 7.8.

Table 7.8 $LALR^1(2)$ **induces** Relation for Partial $LALR(2)$ Machine For Grammar G7.3

Action Table		
State	Lookahead $\tau_1, \tau_2 \in T, T$	Action
S_1	$\{a,b\}$, $\{\$\}$ $\{a\}$, $\{a,b\}$	*reduce* $A \rightarrow$ *shift, goto* S_2
S_2	$\{x\}$, $\{a,b,x\}$ $\{a\}$, $\{a,x\}$	*reduce* $A \rightarrow$ *shift, goto* S_2

Lookahead depth two separates the action pair in state S_1 and depth one separates the action pair in state S_2. Figure 7.6 shows a heterogeneous decision state that implements the state S_1's $LALR^1(2)$ decision.

upon $\tau_1 \in \{a,b\}$ and $\tau_2 \in \{\$\}$ *reduce* $A \rightarrow$;
upon $\tau_1 = a$ and $\tau_2 \in \{a,b\}$ *shift, goto* S_2;

Figure 7.6 Heterogeneous Automaton State for State S_1 of Figure 7.5

A conventional lookahead decision state for S_1 would need space for four 2-tuples whereas the $LALR^1(2)$ state requires space for only three bit sets and a word, which yields a reduction in space from 8 words to about 2. The decision state for S_2 can be implemented as two terminal comparisons rather than the conventional 5 2-tuple compares.

$LALR(2)$ is the dual of $LALL(2)$ and can similarly use the linear approximation $LALR^1(2)$ when analyzing grammars. In addition, the decisions in states S_1 and S_2 are $LALR^1(2)$ resulting in very small space requirements; the $SLR(2)$ version was not also $SLR^1(2)$, but still was able to take advantage of the Λ_i sets to construct small hybrid states. Because grammar G7.3 is $LALR^1(k)$, it is also $LR^1(2)$ as we shall see in the next section.

7.5 $LR(k)$

The $LR(k)$ parsing method has little to gain from the linear approximation analysis as the number of parser states is exponential and full k-lookahead info must be moved along during state construction in case it is needed. However, any **induces** relation (decision state) may utilize linear approximation compression even if compressed analysis is not performed; see Sections 7.20 and 6.18 for a description of how **induces** relations can be compressed. We conclude that $LR^1(k)$ is useful only as a state compression technique and cannot be used to reduce the exponentially complex state construction algorithm. For completeness, this sections provides the $LR(2)$ machine analog of the $SLR(2)$ and $LALR(2)$ machines of previous sections for Grammar G7.3.

Splitting state S_2 of Figure 7.5 results in the $LR(2)$ machine as shown in Figure 7.7.

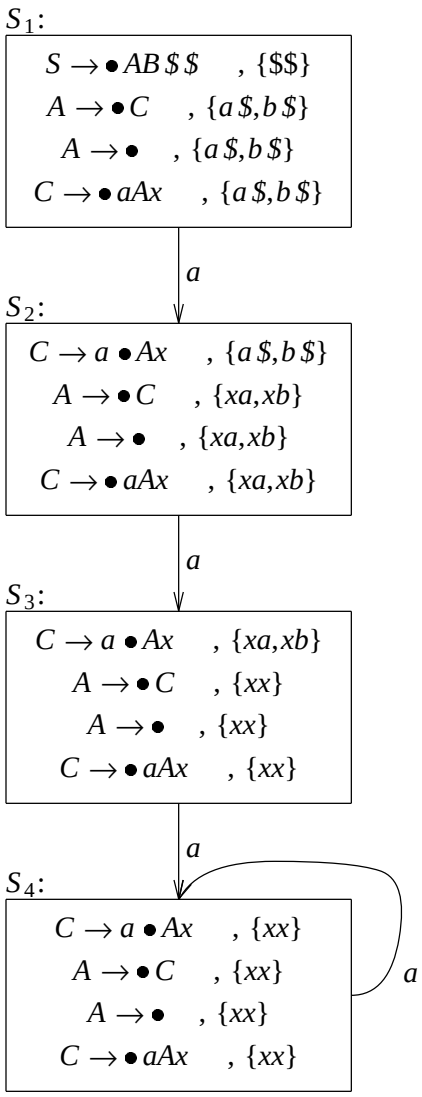


Figure 7.7 Partial *LR*(2) Machine for Grammar G7.3

The $LR(2)$ machine clearly points out that the amount of lookahead information decreases as we progress from $SLR(2)$ to $LALR(2)$ to $LR(2)$ while the number of states increases quickly; $LR^1(2)$ analysis is not possible, but the associated compression can still be used to reduce decision state complexity. States S_2, S_3, and S_4 are all $LR(1)$ decisions and are, therefore, efficient. State S_1 is $LR(2)$ and uses the S_1 **induces** relations in Tables 7.7 and 7.8.

Full $LR(k)$ lookahead sets must be moved along from state to state during $LR(k)$ machine construction; in addition, $LR(k)$ parsers have an exponential number of states. Hence, $LR(k)$ decision states may take advantage of the linear compression, $LR^1(k)$, but cannot use $LR^1(k)$ compression during analysis.

To conclude our discussion of the $LL(k)$ and $LR(k)$ variants, we observe the following: Once an induces relation has been established through analysis of any of the *LL* or *LR* variants, the linear approximation scheme may used to reduce decision state size. For all but the exponentially-large, full $LL(k)$ and $LR(k)$ schemes, linear approximation can also be used to reduce analysis time. More specifically, $LALL^1(k)$, $LL^1(k)$, $SLR^1(k)$, $LALR^1(k)$, and $LR^1(k)$ parsers are well-defined, but $LL^1(k)$ and $LR^1(k)$ lookahead set construction algorithms are not.

The $LL^1(k)$ and $LR^1(k)$-based parsers are attractive due to their linear decision state complexity, but are weaker than full $LL(k)$ and $LR(k)$. The next section generalizes lookahead decisions to use anything from 1-tuple (set) to *k*-tuple comparisons.

7.6 $LL^m(k)$ and $LR^m(k)$

The $LL^1(k)$ and $LR^1(k)$ variants have linear decision state complexity and, for all but full $LL^1(k)$ and $LR^1(k)$, have linear grammar analysis complexity. Because of superior recognition strength, $LL(k)$ and $LR(k)$ variants were also considered in previous sections. We showed how the linear analysis could be used to reduce both grammar analysis time and parser decision state complexity for these parsing strategies. $LL^1(k)$ and $LR^1(k)$ are, for this reason, extremely useful grammar classes. However, some **induces** relations (decision states) cannot be mapped correctly using the linear class and yield large Δ_k sets using the hybrid linear/*k*-tuple mapping. This section generalizes parsing decisions so that the class of **induces** relations, between the linear and *k*-tuple mappings, can be described and decision state complexity can be reduced further.

Define an $LL^m(k)$ parser as a normal, top-down, *LL* parser whose most complex **induces** relation has *m*-tuples as the largest atomic unit and looks no more than *k* terminals ahead for $m \le k$; $LR^m(k)$ parsers are defined similarly. $LL^k(k)$ and $LR^k(k)$ are, therefore, the familiar $LL(k)$ and $LR(k)$. Contrary to the $LL^1(k)$ and $LR^1(k)$ variants, we do not define $LL^m(k)$ and $LR^m(k)$ lookahead computations. These grammar class generalizations are intended to describe more precisely the complexity of an **induces** relation; hence, we limit our discussion to the mapping of **induces** relations.

An example can best illustrate these generalized decisions; in the notation of [SiS90], $C(k)$ represents some class of deterministic parser that we augment to form $C^m(k)$, which represents the same class, but with generalized decisions. We will start with the results of the $C^3(3)$, or $C(3)$, analysis for some decision state and gradually reduce the space complexity of the mapping using $C^m(3)$ techniques; see Table 7.9.

Table 7.9 $C(3)$ **induces** Relation

Lookahead $(\tau_1,\tau_2,\tau_3) \in T^3$	Action
(a,b,c)	1
(c,e,g)	1
(x,y,z)	1
(d,b,c)	2
(a,e,g)	2

For this discussion, will assume that a terminal resides in a full hardware storage word and that terminal sets are encoded as bit sets requiring $|T|/wordsize$ words (about a word in our example). We discuss decision state space complexity because it is important in and of itself and time complexity will generally depend on how much information must be searched to make a parser transition. For example, to implement the $C(3)$ mapping to action 1, three 3-tuples or 9 words are required to store the lookahead information; for action 2, 6 words are needed, which yields a total of 15. In an effort to reduce this complexity, we attempt the minimal, linear $C^1(3)$ mapping (which would yield 6 sets/words). The $C^1(3)$ relation associated with Table 7.9 is shown in Table 7.10.

Table 7.10 $C^1(3)$ **induces** Relation

Lookahead $\tau_1, \tau_2, \tau_3 \in T,T,T$	Action
$\{a,c,x\},\{b,e,y\},\{c,g,z\}$	1
$\{a,d\},\{b,e\},\{c,f\}$	2

The $C^1(3)$ relation is inconsistent because there is no lookahead depth that separates the actions. However, this linear approximation can be used in conjunction the $C(3)$ to form a hybrid $C^1(3)/C(3)$ mapping of the form:

$\tau_1 \in \{a,c,x\}$ *and* $\tau_2 \in \{b,e,y\}$ *and* $\tau_3 \in \{c,g,z\}$ *and* $(\tau_1, \tau_2, \tau_3) \notin (a,e,g)$ *induces* 1
$\tau_1 \in \{a,d\}$ *and* $\tau_2 \in \{b,e\}$ *and* $\tau_3 \in \{c,f\}$ *and* $(\tau_1, \tau_2, \tau_3) \notin (a,b,c)$ *induces* 2

Both induction expressions require 6 words for a total of 12 words. This hybrid approach reduced the number of comparisons over the pure 3-tuple method from 15 to 12 words. More compression can be done by considering $C^2(3)$ information such as that presented in Table 7.11.

Table 7.11 $C^2(3)$ **induces** Relation

Lookahead $(\tau_1,\tau_3) \in T^2$	Action
(a,c)	1
(c,g)	1
(x,z)	1
(d,c)	2
(a,g)	2

By ignoring the terminals appearing at lookahead depth two, the dimension of domain has been reduced. A straightforward collection of 2-tuples yields 10 words — a reduction by 2 over the hybrid $C^1(3)/C(3)$ mapping. Linear compression, $C^1(3)$ can be applied in this case as well to further reduce the complexity. The $C^1(3)$ information is given in Table 7.12.

Table 7.12 $C^1(3)$ **induces** Relation for $C^2(3)$ Information

Lookahead $\tau_1, \tau_3 \in T,T$	Action
$\{a,c,x\},\{c,g,z\}$	1
$\{a,d\},\{c,f\}$	2

Notice, that the ‘‘3’’ in the $C^1(3)$ notation is the maximum lookahead, not how many terminals are examined. Again, the $C^1(3)$ relation is inconsistent, but can be used in a hybrid $C^1(3)/C^2(3)$ mapping; i.e.

$\tau_1 \in \{a,c,x\}$ *and* $\tau_3 \in \{c,g,z\}$ *and* $(\tau_1, \tau_3) \notin (a,g)$ *induces* 1
$\tau_1 \in \{a,d\}$ *and* $\tau_3 \in \{c,f\}$ *and* $(\tau_1, \tau_3) \notin (a,c)$ *induces* 2

This mapping is the least complex at 8 words, using the straightforward tuple comparisons method used here. To reiterate, $C(3)$ information alone requires 15 words, but can be reduced to 12 using the $C^1(3)$ information. The $C^2(3)$ decision class allowed the complexity to be further reduced to 8 words — a significant compression even for this small, contrived example.

Using an exhaustive search by terminal comparison approach, the time complexity for a decision state will be the same as the space complexity. However, using a perfect hash function approach, for example, the **induces** for full $C(3)$ would only require a time complexity of 3 terminal examinations (to compute the hash code from the key). However, nothing would have been done to reduce the space complexity, which is an exponential function in k — here, it is $|T|^3$. As a $C^2(3)$ mapping, time complexity would be 2 and space complexity would be reduced to $|T|^2$. This demonstrates that the $C^m(k)$ generalization can reduce the complexity of many different decision types.

In this section, we generalized $LL(k)$ and $LR(k)$ parsers and parsing decisions to $LL^m(k)$ and $LR^m(k)$, which characterize more precisely the complexity of inducing a parser action given a lookahead string without regards for the parsing method. No new lookahead computations were defined because these generalizations refer specifically to the implementation of **induces** relation mappings. By modulating the values of m and k, the minimum complexity for an **induces** relation implementation can be found for a given decision type.

This chapter explored the variants of $LL(k)$ and outlined how $LR(k)$, and its variants, can take advantage of the techniques emphasized in the previous chapters on $SLL^1(k)$ and $SLL(k)$. We described $LALL(k)$ lookahead information in detail and illustrated the difference between all the $LL(k)$ variants. The effect of linear compression upon $SLR(k)$, $LALR(k)$, and $LR(k)$ parsers and lookahead computations was outlined; the results are very similar to those of the LL variants because, from a decision state implementation point of view, the differences between parsing methods disappear. We observed that $LOOK_k^1$ computations are valuable for all but the full $LL(k)$ and $LR(k)$ strategies, but that the associated linear compression can be applied to any **induces** relation (decision state). Hence, grammar analysis time can be reduced for the non-exponentially large deterministic parsers, but can only reduce decision state complexity for the exponential $LL(k)$ and $LR(k)$ schemes.

Further, in this chapter, we generalized these $LL(k)$ and $LR(k)$ parsers and parsing decisions to $LL^m(k)$ and $LR^m(k)$, which provide a more accurate description of an **induces** relation implementation complexity; specifically, $m \leq k$ is the size of the largest tuple comparison and k is the maximum lookahead depth. We demonstrated how various values of m can be employed to significantly reduce parser decision complexity.

CHAPTER 8 CONCLUSION

Conventional deterministic parsing with lookahead depths greater than one is intractable because lookahead information is potentially exponential; $LL(k)$ and $LR(k)$ parsers also have an exponential number of states, but any of the weaker variants such as $SLL(k)$, $LALL(k)$, $SLR(k)$, and $LALR(k)$ can be used to avoid this problem. Lookahead was previously employed in a straightforward manner — each state transition was a function of the current parser state and the next k terminals of input regardless of whether all k terminals were needed and whether lookahead was needed at all. As a result, each input symbol was inspected exactly k times. The fact that decisions rarely need all k symbols led us to the concept that a new type of parser, called an optimal parser, could be constructed that inspected each input symbol at most once. Further, if each symbol is to be examined at most once, the conventional lookahead atomic unit, the k-tuple, must be dissolved into its constituent components: the individual terminals themselves. By varying the lookahead depth and by allowing non-k-tuple lookahead comparisons, we have removed the two implicit assumptions that led most researchers to consider parsing, for $k>1$, impractical.

The most important contribution of this thesis is the compression of exponential lookahead information to a practical size, which was made possible only by the dissolution of the atomic k-tuple. While others have considered modulating k and examining terminals individually, their parser lookahead decisions are still exponentially large for $k>1$. We introduced linear approximations to full lookahead decisions, called $C^1(k)$, that use lookahead depths up to k, but consider 1-tuples (sets) the largest atomic unit; these decisions have lookahead of size $O(|T| \times k)$ rather than $O(|T|^k)$. Moreover, these approximations are sufficient for most lookahead decisions; e.g., the empirical results of Section 5.11.2 indicate that $SLL^1(k)$ covers about 75% of all $SLL(k)$ decisions for $k>1$. We generalized these approximations to $C^m(k)$, which consider the largest atomic unit to be an m-tuple composed of terminals at contiguous and noncontiguous lookahead depths. When $C^m(k)$ (for $1 \le m \le k-1$) decisions are insufficient, $C(k)$ decisions must be constructed. In this case, full lookahead information also can be compressed heavily. By building hybrid $C^1(k)...C^m(k)/C(k)$ decisions, the typical lookahead $C(k)$ decision can be represented in a practical amount of space. $C^1(k)$ lookahead information may be obtained by compressing the full lookahead information or may be sometimes obtained by computing it directly from the grammar (only full $LL(k)$ and $LR(k)$ cannot use this approach). To that end, we defined a compressed lookahead computation, $LOOK_k^1$, and provided an efficient algorithm that has linear time and space complexity for a fixed grammar — $O(|G| \times k)$. Without $C^m(k)$ decisions, deterministic parsing for $k>1$ would remain infeasible.

Because most work in parsing is theoretical for lookahead depths greater than one, few practical algorithms and data structures existed. Consequently, in this thesis, we provided efficient structures for representing grammars, lookahead, and parsers with heterogeneous states. We introduced new algorithms for computing lookahead, testing for grammar properties, and constructing parser lookahead decisions.

To summarize our approach, recall that we represent grammars as GLA's, which realize a covering, regular approximation to the underlying context-free language. The lookahead sequences of depth k for a position in the grammar correspond to a subset of the sequences of non-ε edges along the walks of length k starting from the associated GLA state. We store the edges found along the walks of the GLA as child-sibling trees, but often view them as lookahead DFA's. Lookahead computations for any $LL(k)$ or $LR(k)$ variant are similar to NFA to DFA conversions.

Unfortunately, obvious algorithms for computing lookahead from GLA's have time and space complexities that are exponential functions of k. We overcome this intractability in three ways: First, the lookahead depth, k, is modulated according the actual requirements of the parsing decision. Second, the linear approximation lookahead is used in place of the normal lookahead when possible. Third, the results of lookahead computations are cached in order to avoid redundant computations.

Although the various $LL(k)$- and $LR(k)$-based parsers need lookahead of different depths for different grammars and grammar positions, lookahead decisions are identical in nature. Each decision is a mapping from a domain of terminals or terminal sequences to a range of parser actions. We abstract the notion of a lookahead decision to a relation called **induces** that describes this mapping; thus, any transformation or implementation of an **induces** relation is equally valid for any parsing strategy and isolates the computation of lookahead from the induction of parser actions and the types of actions. Testing for parser determinism is accomplished by ensuring that the **induces** relations in all parser states are deterministic.

While $LL(k)$ and $LR(k)$ parser construction is well understood from a theoretical standpoint, little practical work has been done because the implementation of lookahead decisions was previously considered intractable. We concentrated, therefore, on the implementation of parser lookahead-decisions. While the worst-case decision size is proportional to the worst-case size of the lookahead information, $O(|T|^k)$, in general, much can be done to reduce this to a practical size. As with lookahead computations themselves, we applied a hierarchical scheme: First, the lookahead depth, k, is modulated to use minimum lookahead. Second, the linear approximation lookahead is employed before full, exponential, k-tuple lookahead. Finally, when the linear approximation is insufficient, a hybrid state composed of the linear approximation plus a set of k-tuples is used. By constructing parsers that use different lookahead depths and comparison structures, parsers with large lookahead buffers become practical. Again, this scheme is only possible by constructing heterogeneous parsers. Chapters 3 and 4 provided a new perspective on lookahead information, lookahead computations, and grammar analysis. Chapter 5 provided a

complete description of $SLL^1(k)$ parsers while Chapter 6 considered full $SLL(k)$ parsers. The $LL(k)$- and $LR(k)$-based parsers were considered in Chapter 7.

LIST OF REFERENCES

[AhU72] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation and Compiling Volume I, Prentice-Hall, 1972.

[AhU73] A.V. Aho and J.D. Ullman, ‘‘A technique for speeding up *LR*(*k*) Parsers,’’ SIAM J. Computing, Vol. 2, No. 2, 1973, pp 106-127.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[ADG91] M. Ancona, G. Dodero, V. Gianuzzi, and M. Morgavi, ‘‘Efficient Construction of *LR*(*k*) States and Tables,’’ ACM TOPLAS Vol. 13, No. 1, January 1991, pp 150-178.

[BeS86] Manuel E. Bermudez and Karl M. Schimpf, ‘‘A Practical Arbitrary Look-ahead LR Parsing Technique,’’ Proceedings of the 1986 Symposium on Compiler Construction (SIGPLAN Notices V21, #7 July 1986), pp 136-144.

[BeS90] Manuel E. Bermudez and Karl M. Schimpf, ‘‘Practical Arbitrary Lookahead LR Parsing,’’ Journal of Computer and System Sciences 41, 1990, pp 230-250.

[Bro74] B.M. Brosgol, ‘‘Deterministic Translation Grammars,’’ TR-3-74, Center for Research in Computer Technology, Harvard University, 1974.

[CuC73] Karel Culik II, and Rina Cohen, ‘‘*LR*-Regular Grammars — an Extension of *LR*(*k*) Grammars,’’ Journal of Computer and System Sciences 7, 1973, pp 66-96.

[DeM75] A.J. DeMers, ‘‘Elimination of Single Productions and Merging Nonterminal Symbols of *LR*(1) grammars,’’ Computer Languages, Vol. 1, 1975, Pergamon Press, Northern Ireland, pp 105-119.

[DeP82] Frank DeRemer and Thomas Pennello, ‘‘Efficient Computation of LALR(1) Look-Ahead Sets,’’ ACM TOPLAS Vol. 4, No. 4, October 1982, pp 615-649.

[DeR69] Frank DeRemer, ‘‘Practical Translators for *LR*(*k*) Languages,’’ PhD Thesis, Department of MIT, Cambridge Massachusetts, 1969.

[DeR71] Frank DeRemer, ‘‘Simple *LR*(*k*) Grammars,’’ Communications of the ACM, Vol. 14, No. 7, 1971, pp 453-460.

[Dob91] H. Dobler, ‘‘Top-Down Parsing in Coco-2,’’ ACM SIGPLAN Notices, Vol. 26, No. 3, March 1991.

[DoP90] H. Dobler and K. Pirklbauer, ‘‘Coco-2 A New Compiler Compiler,’’ ACM SIGPLAN Notices, Vol. 25, No. 5, May 1990.

[FiL88] Charles N. Fischer and Richard J. LeBlanc, Crafting a Compiler, Benjamin/Cummings Publishing Company, 1988.

[Fri79] D. Friede, ‘‘Partitioned *LL* (*k*) Grammars,’’ Automata, Languages and Programming. Lecture Notes in Computer Science 71, Springer Verlag, 1979, pp 245-255.

[Gre65] Greibach, ‘‘A New Normal Form Theorem for Context-Free Phrase Structure Grammars,’’ Journal of ACM 12, 1965, pp 42-52.

[HSU75] Harry Hunt, Thomas Szymanski and Jeffrey D. Ullman, ‘‘On the Complexity of *LR* (*k*) Testing,’’ Communications of ACM 18, 1975, pp 707-716.

[HuS78] Harry Hunt and Thomas Szymanski, ‘‘Lower Bounds and Reductions Between Grammar Problems,’’ Journal of the ACM 25, No. 1 January 1978, pp 32-51.

[Ive86] Fred Ives, ‘‘Unifying View of Recent LALR(1) Lookahead Set Algorithms,’’ Proceedings of the 1986 Symposium on Compiler Construction (SIGPLAN Notices V21, #7 July 1986), pp 131-135.

[Joh78] Stephen Johnson, ‘‘Yacc: Yet Another Compiler-Compiler,’’ Bell Laboratories, Murray Hill, NJ, 1978.

[JoS75] Donald Johnson and Ravi Sethi, ‘‘Efficient Construction of LL(1) Parsers,’’ Pennsylvania State University Computer Science TR 164, 1975.

[Knu65] Donald Knuth, ‘‘On the Translation of Languages from Left to Right,’’ Information and Control 8, 1965, pp 607-639.

[Knu71] Donald Knuth, ‘‘Top-Down Syntax Analysis,’’ Acta Informatica 1, 79-110, 1971.

[KrM81] Bent Bruun Kristensen and Ole Lehrmann Madsen, ‘‘Methods for Computing *LALR* (*k*) Lookahead,’’ ACM TOPLAS, Vol. 3, No. 1, January 1981, pp 60-82.

[LaL76] Wilf R. LaLonde, ‘‘On Directly Constructing *LR* (*k*) Parsers Without Chain Reductions, 3rd ACM Symposium on Principles of Programming Languages, 1976, pp 127-133.

[LeS68] P.M. Lewis II and R.E. Stearns, ‘‘Syntax-Directed Transduction,’’ Journal of the ACM, Vol 15, No. 3, 1968, pp 465-488.

[PDC92] T.J. Parr, H.G. Dietz, and W.E. Cohen, ‘‘PCCTS Reference Manual Version 1.00,’’ ACM SIGPLAN Notices, February 1992.

[Pen86] Thomas Pennello, ‘‘Very Fast LR Parsing,’’ Proceedings of the 1986 Symposium on Compiler Construction (SIGPLAN Notices V21, #7 July 1986), pp 145-151.

[Rob90] George H. Roberts, ‘‘From Recursive Ascent to Recursive Descent: Via Compiler Optimizations,’’ SIGPLAN Notices, Vol. 25, No. 4, April 1990.

[RoS70] D.J. Rosendrantz and R.E. Stearns, ‘‘Properties of Deterministic Top-Down Grammars,’’ Information and Control 17, 1970, pp 226-256.

[SiS82] Seppo Sippu and Eljas Soisalon-Soininen, ‘‘On LL(k) Parsing,’’ Journal of Information and Control, Vol 53, 1982. pp 141-164.

[SiS83] Seppo Sippu and Eljas Soisalon-Soininen, ‘‘On the Complexity of LL(k) Testing,’’ Journal of Computer and System Sciences, Vol 26, 1983. pp 244-268.

[SiS88] S. Sippu and Eljas Soisalon-Soininen, ‘‘Parsing Theory Volumne I,’’ Springer Verlag, Berlin, 1988.

[SiS90] S. Sippu and Eljas Soisalon-Soininen, ‘‘Parsing Theory Volumne II,’’ Springer Verlag, Berlin, 1990.

[Ukk83] Esko Ukkonen, ‘‘Lower Bounds on the Size of Deterministic Parsers,’’ Journal of Computer and System Sciences, Vol 26, 1983. pp 153-170.

APPENDIX

Sample Grammar Submissions From PCCTS Users

This appendix gives a description of the grammars submitted by users of PCCTS, the Purdue Compiler-Construction Tool set, following a request for such. The grammars were stripped of actions, converted to BNF notation. In addition, to protect the privacy of the original grammars, the users converted all terminal and nonterminals to generic character strings. The grammars were examined by the algorithms presented in this thesis.

[1] **Prototype C compiler front end**; Peter Dahl (dahl@everest.ee.umn.edu), HPC Graduate Fellow, Army High Performance Computing Research Center.

[2] **Command parser for a Nonlinear Finite Element Analysis Software**; Tom Zougas (zougas@me.utoronto.ca)

[3] **Fortranp grammar — Serial to Parallel Fortran Translator**; Matthew O'Keefe, Terence Parr, B. Kevin Edgar, Steve Anderson, Paul Woodward, and Hank Dietz.

[4] **ANSI C grammar distributed with PCCTS 1.06**; Terence Parr (parrt@ecn.purdue.edu).

[5] **Pascal Grammar distributed with PCCTS 1.06**; Will Cohen (cohenw@ecn.purdue.edu) and Terence Parr (parrt@ecn.purdue.edu).

[6] **Advanced tutorial string C from PCCTS 1.06**; Terence Parr (parrt@ecn.purdue.edu).

[7] **ANTLR language description (antlr.g) PCCTS 1.06**; Terence Parr (parrt@ecn.purdue.edu).

[8] **DLG language description (dlg_p.g) PCCTS 1.06**; Will Cohen (cohenw@ecn.purdue.edu).

[9] **Front end to TROFF that makes it smell less bad; this thesis is written using it**; Terence Parr (parrt@ecn.purdue.edu).

[10] **Converts LISP tree notation to EQN/PIC graphics**; Terence Parr (parrt@ecn.purdue.edu).

[11] **Converts an NFA description to PIC code**; Terence Parr (parrt@ecn.purdue.edu).

[12] **Accepts BNF and puts into data structure**; Terence Parr (parrt@ecn.purdue.edu).

[13] **EBNF->BNF converter**; Terence Parr (parrt@ecn.purdue.edu).

[14] **Compiler for a parallel programming (ELP)**; Mark Nichols, Gene Saghi, Dan Watson, Mu-Cheng Wang, Robert Palmer, H.J. Siegel, and Hank Dietz.

[15] **Grammar for a case tool language. Currently our case tool is lisp based with an LALR parser. We have converted our grammar over to PCCTS**; Frank Korzeniewski (frkorze@pacbell.com).

[16] **COBOL - WSL translator**; Gareth L de C Morgan (g.l.morgan@durham.ac.uk), University of Durham.

[17] **Grammar used for parsing small modula-2 subset (currently only brief expressions and conditionals.)**; Tom Rushton (T.G.A.Rushton@durham.ac.uk).

[18] **Grammar used for parsing large quantities of Modula-2. It won't handle non-simple types, and imported functions**; Tom Rushton (T.G.A.Rushton@durham.ac.uk).

[19] **Ada-like "Macro" student project grammar from "Crafting a Compiler in C" by Fischer & LeBlanc, used for graduate level compiler writing course**; Roy B. Levow (roy@gemini.cse.fau.edu), Florida Atlantic University.

[20] **Parses a wide range of SQL SELECT syntax**; Fred Scholldorf (scholldorf@nuclear.physics.sunysb.edu)

[21] **MACRO Compiler, A project for Compiler Writing Class**; Michael P. Vogt (mike_vogt@vnet.ibm.com).

[22] **Convert a Linear Programming problem description into Matrix-Vector form**; Gautham Kudva (kudva@ecn.purdue.edu).

VITA

Terence John Parr was born in Los Angeles, California, USA in the year of the dragon on August 17, 1964 during the week of the Tonkin Gulf Crisis, which eventually led us into the Vietnam Conflict; coincidence? Terence's main hobbies in California were drooling, covering his body in mud, and screaming at the top of his lungs.

In 1970, Terence moved to Colorado Springs, Colorado with his family in search of better mud and less smog. His formal education began in a Catholic grade school where he became intimately familiar with penguins and other birds of prey. Terence eventually escaped private school to attend public junior high only to return to the private sector — attending Fountain Valley School for the ''education'' only a prep school can provide. After being turned down by every college he applied to, Terence begged his way into Purdue University's School of Humanities. Much to the suprise of his high school's faculty and the general populace, Terence graduated in 1987 from Purdue with a bachelor's degree in computer science.

After contemplating an existence where he had to get up and go to work, Terence quickly applied to graduate school at Purdue University's School of Electrical Engineering. By sheer tenacity, he was accepted and then promptly ran off to Paris, France after only one semester of graduate work. Terence returned to Purdue in the Fall of 1988, eventually finishing up his master's degree in May 1990 despite his best efforts. Hank Dietz served as major professor and supervised Terence's master's thesis.

A short stint with the folks in blue suits during the summer of 1990, convinced Terence to begin his Ph.D.; again, Hank Dietz was his advisor. He passed the Ph.D. qualifier exam in January of 1991, stunning the local academic community. After three years of course work, research, and general fooling around, Terence finished writing his doctoral dissertation and defended it against a small horde of professors and students on July 1, 1993.

LIST OF REFERENCES

APPENDIX

VITA