THE IDENTIFICATION AND ELIMINATION OF REDUNDANT ACTIONS AND STATES FROM NONCANONICAL LR PARSERS

BY

LE, WEIJUN VIVI

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba

ISBN 0-315-76914-9

THE IDENTIFICATION AND ELIMINATION OF REDUNDANT ACTIONS AND STATES FROM NONCANONICAL LR PARSERS

BY

WEIJUN VIVI LE

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Contents

Abstract

The noncanonical SLR(1) parsing method is a two-stack extension of the SLR(1) parsing method that works for a larger class of grammars and languages. Existing NSLR(1) construction algorithms generate some useless parser actions and states.

In this thesis, several functions and relations on NSLR(1) parsers are defined and analyzed. The analysis leads to two algorithms of different complexity, which can detect and delete useless parser actions and redundant states. A modified NSLR(1) parser construction algorithm is proposed that can lead to even smaller parser than those generated by existing methods.

Key words and phrases: Context-free grammar, SLR parsing, noncanonical parsing.

List of Figures

List of Tables

Chapter 1

Introduction

If you look inside the language reference manuals for most of the modern programming languages, such as Pascal, Modula-2, Modula-3, Ada, and Oberon, you will find a character-level grammar for that language. A character-level grammar is a grammar that uses only single characters as the terminal symbols. Thus all symbols, such as identifiers, constants, and keywords, are described right down to their constituent characters. Character-level grammars attempt to give a full description of a language; avoiding the need for supplementary English descriptions.

Despite the popularity of character-level grammars, none of the commonly used parser generation techniques, including LL(1), SLR(1), and LALR(1) parser generators, are powerful enough to handle such grammars as published. A single character of lookahead, one terminal symbol, is not enough to resolve parsing conflicts. This deficiency is usually handled by partitioning the language recognizer into two phases, a scanner phase and a parser phase, and by partioning the grammar into two parts, one part for each phase. Such a solution to the problem has several common drawbacks:

1) The partitioned grammar is usually described in two different metalanguages: regular expressions for the scanner phase, and a context-free grammar for the parser phase.

2) The partitioned grammar is usually processed by two different parser generators: a scanner generator (often a human programmer) and a parser generator.

3) The interface between the two modules is complex, involving both symbol codes, and semantic values of different forms (identifiers, numeric values, and literal character strings). In other words the two modules are strongly coupled.

4) The cohesion of the scanner module is quite poor. It can be characterized as having logical cohesion; the cohesion of tasks that are similar in nature, but otherwise unrelated. This is reflected by the fact that there is little common code used in the recognition of identifiers, numeric constants, and string constants.

5) The technique works poorly for source-to-source translators that do not wish to delete white space and comments from the input.

In order to be able to process character-level grammars as published, more powerful practical parser generation techniques are needed. In addition it would be desirable to handle these grammars without sacrificing the linear-time performance of the existing popular parsing techniques. In 1989, Salomon and Cormack [Sal89B] proposed that noncanonical SLR(1) parsing, a method invented by K.C. Tai [Tai79], can be used for this task.

A bottom-up parsing technique that can make nonleftmost possible reductions in sentential forms is said to be *noncanonical*. One noncanonical extension, the one used by Tai [Tai79], would be to allow a parser to perform shifting and reduction of right context, and to use the resulting nonterminal symbols as lookahead characters for a temporarily postponed action. Nearly every existing parsing technique can be extended in this way to

become a noncanonical method which operates on a larger class of grammars and languages than the original technique. With this processing of right-context a scanner phase becomes unnecessary.

Existing noncanonical parser construction techniques, which function by the expansion of corresponding canonical parsers, generate useless lookahead symbols, unreachable transitions as well as unreachable states. Some transitions and states will never be used during parsing because of the transitions and states added for noncanonical parsing. Some lookahead symbols may be useless for the same reason and because of the addition of too many nonterminals to the lookahead sets. Tai mentions this fact but presents no algorithms for eliminating useless actions and states. The problem of the automatic optimization of noncanonical parsers is discussed in this thesis. New relations, some of which extend those defined by DeRemer [DeRe82], are defined here and are analyzed to capture the essential properties of noncanonical automata. General methods of detecting useless lookahead symbols and unreachable transitions are proposed. Two algorithms with different levels of optimization were implemented for *SOAP*, an NSLR(1) parser generator.

This thesis is organized as follows. Chapter 2 is a summary of the basic terminology used throughout this thesis. In Chapter 3, a survey of some previous work on noncanonical parsing is presented. The basic ideas of noncanonical parsers are introduced. Several examples are presented to clarify the differences between noncanonical and canonical parsers, and to demonstrate the existence of useless shift and reduce actions in the action tables generated by current NSLR(1) construction algorithm. In Chapter 4, relations and functions to be used to explore the essential properties of noncanonical SLR(1) parsers are defined. These relations and functions are used to ensure that the algorithms presented in Chapter 5 are

correct. In Chapter 5, two algorithms are presented. My first approach is to detect useless actions by building relations on nonterminal symbols. The reason that this simple solution fails to detect some useless actions is explained. A more powerful algorithm is proposed by building relations on states. In Chapter 5, we also present the experimental results of the two algorithms on grammars for PASCAL, Modula 2, and some grammars provided by Kuo-Chung Tai [Tai79]. The results are not encouraging for character level grammars for real programming languages. The reason that such parsers cannot be improved is analyzed. The analysis leads to the invention of a modified parser construction algorithm, that does permit significant shrinking of parser size.

Chapter 2

The Notation of Canonical Parsing

In this chapter, the notation used in the rest of this thesis is presented. We also briefly introduce SLR parsers, and functions used in LR parser construction algorithms.

2.1 Terminology

First, we present the notation system used in this thesis. V_N is a finite set of nonterminal symbols, V_T is a finite set of terminal symbols, and $V = V_N \cup V_T$ is the set of all grammar symbols. The set V^*, the reflexive transitive closure of V, contains all strings composed of symbols in V, including the empty string, which is represented by ϵ.

The following standard convention for the meaning of Roman and Greek letters are used in this thesis.

$$A, B, C, \ldots \in V_N$$

$$\ldots, X, Y, Z \in V$$

$$a, b, c, \ldots \in V_T$$

$$\ldots, x, y, z \in V_T^*$$

$$\alpha, \beta, \gamma, \ldots \in V^*$$

The letter $S \in V_N$ denotes the start symbol of a grammar. Called the length of α, $|\alpha|$, is the number of occurrences of symbols of V in α.

If R is a relation, R^* denotes the reflexive transitive closure of R, and R^+ denotes the transitive closure.

The production set P is a finite subset of $V_N \times V^*$, whose members take the form $A \rightarrow \alpha$, where A is called the *left-part*, and α is called the *right-part.* If $A \rightarrow \alpha$ is a production and $\beta A \gamma$ is a string in V^+, then we write $\beta A \gamma \Rightarrow \beta \alpha \gamma$ and say that $\beta A \gamma$ directly *derives* $\beta \alpha \gamma$. A *sentential form* of G is a string α such that $S \Rightarrow^* \alpha$ and α is in V^*. A *sentence* x of G is a sentential form of G consisting solely of terminals, i.e., x is in V_T^*. The *language* $L(G)$ generated by G is the set of sentences generated by G, i.e., $L(G) = \{x \mid S \Rightarrow^* x\}$.

A derivation in which the rightmost nonterminal is replaced at each step is said to be a rightmost derivation. If $\alpha \Rightarrow \beta$ is a step in which the rightmost nonterminal in α is replaced, we write $\alpha \Rightarrow_{rm} \beta$. Every rightmost step, using our notational conventions, has the form $\gamma A y \Rightarrow \gamma \delta y$ in which y consists of terminals only and $A \rightarrow \delta$ is a production. If $S \Rightarrow_{rm}^* \alpha$, then we say α is a *right-sentential form* of the grammar at hand. Rightmost derivations are also called *canonical* derivations.

2.2 LR parsing

An LR parser, also known as a *bottom-up* or *shift-reduce* parser, operates by scanning an input word from left to right, and constructing the rightmost derivation in reverse. To construct a parser for a grammar G, G is augmented with a new start symbol S', not in V, and a new production $S' \rightarrow S\$$, assuming that every input string is followed by the symbol \$, where \$ is not in V. The new production is assumed to be the zeroth production.

The two basic actions in a parser are to shift an additional character of input onto the state stack, or to reduce a handle on the state stack to the nonterminal symbol which generates it. The parse ends when the state stack contains the start symbol and the input is the termination symbol \$. The rightmost derivation generated by the operation is the *parse* of the input word.

More formally, a canonical LR automaton for a *context-free grammar* (CFG) G, $G = (V_N, V_T, P, S)$, is a sextuple $\mathrm{LRA}(G) = (K, V, P, s_{start}, \mathrm{Action}, \mathrm{NEXT})$, where K is a finite set of states, whose members are represented by a subscripted letter s, such as s_p, s_q, s_1, and s_2. The symbol $s_{start} \in K$ represents the start state. The parsing Action function is a map from $K \times (V_T \cup \{\$\})$ into $\{shift,\ error,\ accept\} \cup \{reduce\ I \mid I \in P\}$. The function NEXT is a map from $K \times V$ into $K \cup \{error\}$, each member is called a transition. The transition (s_q, X) is represented by $s_q \xrightarrow{X} s_p$, where $s_p = \mathrm{NEXT}(s_q, X)$, or by $s_q \xrightarrow{X}$ when s_p is irrelevant, and we call X the *accessing symbol* of state s_p. Each state has a unique accessing symbol, except s_{start}, which has none. In this thesis we assume that the grammar has no duplicate or useless productions, and no useless symbols.

Function PRED, which is the inverse of NEXT, takes a state s_p as an argument, and

produces a set of states as output. State s_q is in PRED(s_p) if $s_q \xrightarrow{Y} s_p$, where Y is the accessing symbol of state s_p.

$$\text{PRED}(s_p) = \{s_q \mid s_q \xrightarrow{Y} s_p, Y \in V\}$$

A path H is a sequence of states $s_{q0}, \cdots, s_{qn}$ such that

$$s_{q0} \xrightarrow{X_1} s_{q1} \rightarrow \cdots \rightarrow s_{q(n-1)} \xrightarrow{X_n} s_{qn}.$$

The path H is denoted by $[s_{q0} : \alpha]$, where $\alpha = X_1, \cdots, X_n$, and $\text{Top}(H) = s_{qn}$. The concatenation of $[s_q : \alpha]$ and $[s_p : \beta]$, where $\text{Top}([s_q : \alpha]) = s_p$, is written $[s_q : \alpha]\ [s_p : \beta]$ and denotes $[s_q : \alpha\beta]$. $[s_{start} : \alpha]$ can be abbreviated $[\alpha]$; thus [ ] denotes s_{start} alone. We say that α accesses s_q if Top($[\alpha]$)= s_q.

A *configuration* is a member of $K^+ \times V_T^+$ and can be presented by $[\alpha]\beta$, its first part is a state accessed by α, and its second the unprocessed input β. The relation $\vdash$ on the configuration represents the next move of the parser and is the union of $\vdash_{shift}$ and $\vdash_{A \rightarrow \omega}$, for all $A \rightarrow \omega \in P$.

We denote an item of an LR automaton LRA(G), with respect to a production $A \rightarrow \alpha$ as $[A \rightarrow \beta \bullet \gamma, y]$, where $\alpha = \beta\gamma$. Each item represents a partition of the right hand side of a production. Informally, the item represents a production which is *partially recognized*, so we call items as above where $\alpha = \beta$, (i.e. $\gamma = \epsilon$) complete items, denoted $[A \rightarrow \alpha\bullet, y]$. The fixed-sized string of lookahead symbols y denotes possible terminal symbols that can follow the nonterminal A of this item.

LRA(G) states are elements of the power set of items. There are two types of items in a state, kernel items, which define the state, and closure items, which complete the state, and are derived through a closure operation on the kernel items.

A CFG is LR(1) if a parser exists which can always make the decision to *reject*, *accept*, *shift* or *reduce* (and by which production to *reduce*) with a one symbol of lookahead in the remaining terminal string. Unless otherwise specified, only parsers with $k = 1$ lookahead symbol are discussed in this thesis.

An inadequate state in a parser is one for which the parser cannot deterministically decide what move to make. These fall into two classes, *reduce-reduce* conflicts, and *shift-reduce* conflicts. In the first class, a parser knows there is a handle, but the information on the lookahead stack is not sufficient to determine which production should be used in a reduction. In the second class, the parser cannot tell if the string shifted in is a handle. It might be correct to reduce the string to a nonterminal symbol, or it might be correct to shift more symbols to form a different handle.

2.3 The Sets FIRST, LAST, and FOLLOW

Here we describe functions we use in later discussion.

Let FIRST(α) be the set of terminals and nonterminals that can be the first symbol of any sentential form derivable directly from α.

$$\text{FIRST}(\alpha) = \{Y \mid \alpha \Rightarrow Y\beta\}$$

And let LAST(α) be the set of terminals and nonterminals that can be the last symbol of

any sentential form derivable directly from α.

$$\text{LAST}(\alpha) = \{Y \mid \alpha \Rightarrow \beta Y\}$$

Let $\text{FIRST}^*(\alpha)$ be the reflexive transitive closure, and $\text{FIRST}^+(\alpha)$ be the transitive closure of $\text{FIRST}(\alpha)$, and $\text{LAST}^*(\alpha)$ be the reflexive transitive closure, and $\text{LAST}^+(\alpha)$ be the transitive closure of $\text{LAST}(\alpha)$. Thus, for instance, $\text{FIRST}^*(\alpha) = \{Y \mid \alpha \Rightarrow^* Y\beta\}$. The function $\text{FIRST}^{-1}(Y)$ is the inverse of function $\text{FIRST}^*(\alpha)$, and $\text{LAST}^{-1}(Y)$, the inverse of $\text{LAST}^*(\alpha)$.

$$\text{FIRST}^{-1}(Y) = \{X \mid Y \in \text{FIRST}^*(X)\}$$

$$\text{LAST}^{-1}(Y) = \{X \mid Y \in \text{LAST}^*(X)\}$$

The symbol B is ranked as a *lower-level* symbol of A if B is in $\text{FIRST}^+(A)$, and A is not in $\text{FIRST}^+(B)$. We say A and B are *same-level* symbols if A is in $\text{FIRST}^+(B)$ and B in $\text{FIRST}^+(A)$. Terminals are always *lowest-level* symbols.

T_FOLLOW(A), for a nonterminal A, is a set of terminals that can follow A in some sentential form, and if A can be the rightmost symbol of a sentential form, then \$ is included in T_FOLLOW(A). That is,

$$\begin{aligned}\text{T\_FOLLOW}(A) \;=\; &\{b \in V_T \cup \{\$\} \mid S' \stackrel{*}{\Rightarrow} \beta A b \gamma, \\ &\quad \text{for some } \beta \text{ in } \text{V}^* \text{ and } \gamma \text{ in } \text{V}^*\$\}\end{aligned}$$

Function FOLLOW(A), for a nonterminal A, is a set of symbols, terminals and nonter-

minals, that can follow A in some sentential form, and if A can be the rightmost symbol of a sentential form, then \$ is included in FOLLOW(A). That is,

$$\begin{aligned} \text{FOLLOW}(A) \;=\; & \{Y \in V \cup \{\$\} \mid S' \stackrel{*}{\Rightarrow} \beta A Y \gamma, \\ & \qquad \text{for some } \beta \text{ in V}^* \text{ and } \gamma \text{ in V}^*\$\} \end{aligned}$$

Thus the function T_FOLLOW(A) can also be defined as T_FOLLOW(A) = $\{\ b \in V_T \cup \{\$\} \mid b \in \text{FOLLOW}(A)\ \}$.

2.4 SLR(1) Parsers

To construct a canonical SLR(1) parser, we first precompute the T_FOLLOW sets as previously defined, then the LR(0) parser is built using the LR construction algorithm with null lookahead sets. Since items of the parser have no lookhead, we will simplify their definition to $[A \rightarrow \beta \bullet \gamma]$.

An LR(0) parser state is inadequate if it contains more than one item, at least one of which is complete, since there is no lookahead to distinguish between multiple actions.

The SLR(1) automaton attempts to resolve the conflicts through the addition of lookahead symbols for complete items. SLR(1) lookahead is computed, for the complete items of each state, as follows:

$$LA(s_q, A \rightarrow \alpha) = \{x \mid x \in \text{T\_FOLLOW}(A)\}$$

The correctness follows because $LA(s_q, A \rightarrow \alpha)$ simply states that parsing can continue

by reducing α to A only if the following symbol in the input stream could possibly arise after A in some derivation from S', which is obviously necessary for the result to be a sentential form deriving a word $x \in L(G)$.

Note that simple LR lookahead for $A \rightarrow \alpha$ is independent of the state in which it is being applied (other than that the state contains the production $A \rightarrow \alpha$). This differs from LR lookahead, which enforces that parsing continue with reduction $A \rightarrow \alpha$ only if the following symbol could possible arise after A in some derivation consistent with the current state of the parsing automaton.

2.5 Summary

In this chapter, we presented the notation and terminology used in the remainder of this thesis, which is largely standard and can be found in such sources as [Aho86], and [DeRe82]. The very few differences in terminology are taken from the Tai in [Tai79], and Salomon in [Sal89A]. We also presented functions which are used in SLR(1) and NSLR(1) construction algorithms and will be used in following chapters. We introduced briefly the idea of SLR(1) parsers. Detailed information about SLR(1), LR(1), LALR(1) parsers can be found in most compiler books like [Aho86].

Chapter 3

Noncanonical Parsing and The Objective of This Research

A survey of noncanonical LR parsers is presented in this chapter. The basic ideas of the NSLR technique and the difference between SLR and NSLR parsers are introduced. In addition, two examples will be given to demonstrate that useless actions can exist in the parsers generated by current noncanonical parser construction techniques.

3.1 Survey

The well known LR(k) parsing algorithm of Knuth [Knuth65] and its two major variations SLR(k) and LALR(k) due to DeRemer [DeRe71] are *canonical* in that they reduce only leftmost phrases of handles, with k terminal lookahead symbols, where k is a constant. Such parsers can be implemented by a single-stack machine with a fixed-size parse table, that is a pushdown automaton.

Compared with canonical parsing, noncanonical parsing allows greater freedom to select a phrase for reduction and therefore has several advantages. First, the set of grammars which are deterministically parsable by noncanonical parsing is larger. Second, the set of languages defined by noncanonically-parsable grammars may contain some nondeterministic languages. Some of the existing canonical parsing methods can be extended for noncanonical parsing without loss of parsing efficiency. A brief discussion of existing noncanonical parsing methods follows.

3.1.1 Noncanonical Extension of Simple Precedence(SP) Parsers

Colmerauer [Col70] defined total precedence relations, which are generalizations of the Wirth and Weber simple precedence relations, such that $<\cdot$ and $\cdot>$ indicate the left and right ends, respectively, of a phrase. By requiring that at most one precedence relation hold between any pair of symbols, at least one phrase of every sentential form is uniquely distinguished by the relations at its left and right ends. Colmerauer showed that the total precedence languages are incommensurate with both the deterministic languages and their reflections.

3.1.2 Noncanonical Extension of Bounded Context(BC) Parsers

Floyd's notion of Bounded Context Parsers [Floy64] provided much of the framework for the discovery of the LR(k) languages. A grammar is said to be m, n *bounded context* (BC(m, n)) if every phrase of any sentential form is uniquely distinguished by the m symbols to its left and the n symbols to its right. If this restriction is weakened by specifying that at least the leftmost phrase has bounded context, then the class, BRC(m, n), of *bounded right*

context grammars, is a proper subclass of the LR(n) grammars, as LR(n) grammars have right context of at most n symbols, and unbounded left context. However, the classes are equivalent when viewed in language space.

BRC parsing is canonical, in the LR sense, as only leftmost phrases are reduced. Williams [Will75] derived a noncanonical extension of the BC method and defined a class of grammars, called the m, n *bounded context parsable* (BCP(m, n)) grammars, in which at least one phrase of any sentential form is uniquely distinguished by the m symbols to its left and the n symbols to its right. Currently, the BCP method is the most powerful of all parsing methods that have both decidability and linear time parsability. However, this method is not practical because it requires that a large table be scanned to distinguish a phrase.

3.1.3 Noncanonical Extensions of LR(k) Parsers

Szymanski and Williams [Syzm76] attempted to construct a general framework for bottom-up parsers and, within that framework, to examine noncanonical extensions of some existing bottom-up parsing methods. One noncanonical extension of LR(k), which was suggested by Knuth [Knuth65] and called LR(k, t), requires that in any sentential form one of the leftmost t phrases be uniquely distinguished by its left context and the first k symbols of the right context. Szymanski and Williams showed that the LR(k, t) grammars are a superset of the LR(k) grammars, but the LR(k, t) languages are exactly the LR(k) languages (i.e. the deterministic languages).

Two other noncanonical extensions of LR(k) were studied by Szymanski and Williams. One of them, FSPA(k), requires that a finite-state parsing automaton be able to find a

phrase in any sentential form by using a left-to-right scan with k symbols of lookahead. The other, which includes FSPA(k) as a proper subclass and is called LR(k, ∞), requires that at least one phrase of every sentential form can be found solely by examining its left context and the first k symbols of its right context. They showed that the question of whether an arbitrary CF grammar is FSPA(k) and LR(k, ∞) for any fixed value of k is undecidable and suggested that further restriction to the FSPA method to achieve decidability should be considered. RNP(k), a noncanonical extension of SLR(k) proposed by Szymanski, is a proper subclass of FSPA(k). It is defined by requiring that phrases which have been scanned be reduced as early as possible.

3.1.4 Noncanonical SLR Parsing

Two noncanonical extensions of the simple LR(1) method are presented by Tai [Tai79]. He defined a class of context-free grammars called leftmost SLR(1) by using lookahead symbols which appear in leftmost derivations. This class includes the SLR(1), reflected SMSP, and total precedence grammars as proper subclasses. Another larger class of context-free grammars is called noncanonical SLR(1), that is NSLR(1) for short. The NSLR(1) languages can be recognized deterministically in linear time using a two-stack pushdown automaton. With no ϵ-productions, the NSLR(1) grammars are exactly the RNP(1) defined by Szymanski. The noncanonical SLR(1) method of Tai was chosen by Salomon [Sal89A] to supply the parsing power needed for character-level grammars.

Tai's parser-generation method cannot be used for character-level grammars exactly as published. It requires several improvements. One error in Tai's method is that lookahead sets for reduce items in unexpanded states contain only terminal symbols. It is possible,

however, that during a parse, an unexpanded state may be presented with a valid nonterminal as a lookahead symbol and as a result the parser would incorrectly terminate with an error. Salomon and Cormack corrected this error by allowing all reduce items in all states, expanded and unexpanded, to have all-possible following symbols in their lookahead sets.

Another deficiency of Tai's algorithm is the handling of *ϵ-productions*. Tai's use of *ϵ-closure* on inadequate parser states, generates more complete items from *ϵ-productions* than are actually needed. These extra items can introduce new conflicts that unnecessarily cause the rejection of some grammars. Salomon proposes a method for reducing the number of *ϵ-reduce* items generated, and in this way admit an important class of practical grammars previously rejected.

3.2 The Noncanonical SLR(1) Parsing Automaton

In this section and the next section, we present a description of NSLR(1) parsers and NSLR(1) parser construction methods. The NSLR(1) construction method begins by constructing SLR(1) parsing tables. Whenever an inadequate item set is reached, the parser postpones any conflicting reductions, and continues parsing the right context in order to obtain a nonterminal to be used as a lookahead symbol. Since further right context is used in the reduction of the new lookahead symbol, the new lookahead symbol can often resolve the original conflict.

An NSLR(1) parser has two stacks, a state stack and a lookahead stack (see Figure 3.1). It uses the four usual parser actions, *shift*, *reduce*, *error*, and *accept*, but two of these actions change slightly. The first change is that the *reduce* action is redefined so that the

left part of the reduce item, instead of being shifted immediately, is pushed back onto the lookahead stack to serve as the lookahead symbol for the next parser action. The top of the lookahead stack contains the lookahead symbol for the parser. If the lookahead stack is empty, the next symbol will be taken from the input stream. The second change is that the *shift* action is allowed to shift nonterminals as well as terminals, in order to accommodate the previous change.

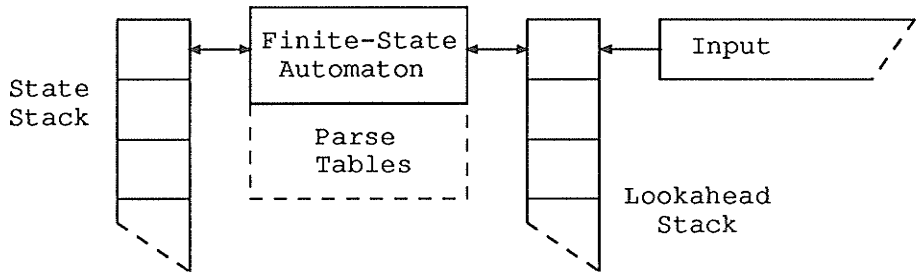

Figure 3.1: The noncanonical LR parsing automaton.

There is a slightly different way, used by Tai, to describe a noncanonical parser. In this second description, a parser is initiated with all the input stacked on the lookahead stack, with the front of the input at top of the stack. This description is functionally identical to the one given above, but it simplifies the specification of the parsing algorithm. In this description, popping the lookahead stack encompasses the operations of popping the lookahead stack and shifting the next input character if the lookahead stack is empty. As a result, we use the second description in the rest of this thesis.

3.3 Extending SLR(1) Parsers to Become NSLR(1) Parsers

In this section, a simple example is used to illuminate the construction of noncanonical SLR(1) parsers. Consider the grammar G_1 with the productions in Figure 3.2:

$$
\begin{aligned}
S &\rightarrow Aa \mid Bb \\
A &\rightarrow EA \mid E \\
B &\rightarrow FB \mid F \\
E &\rightarrow c \\
F &\rightarrow c
\end{aligned}
$$

Figure 3.2: Grammar G_1: G_1 is not LR(k) for any k.

Figure 3.3 shows the SLR(1) parser for G_1 in which s_1 is the only SLR(1) inadequate state. Actually, G_1 is not LR(k) for any k because the nonterminal, E or F, to which c should be reduced depends upon whether the final input symbol is a or b, respectively. Since the input can be of unbounded length, unbounded lookahead would be needed to resolve the conflict.

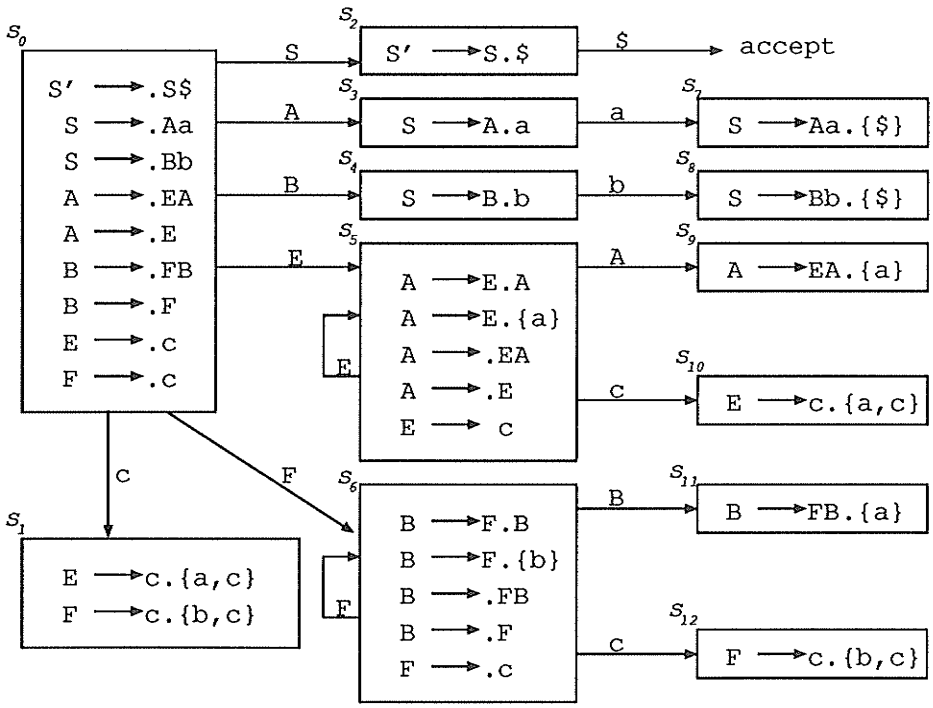

Figure 3.3: SLR(1) parser for G_1.

In order to resolve the conflict, noncanonical parsing can be applied to reduce nonleft-

most phrases of sentential forms. We present a modified NSLR(1) construction algorithm due to Salomon and Cormack [Sal89B], which incorporates several fixes and enchancements to Tai's original algorithm. For a detailed discussion, please see [Tai79], [Sal89B], and [Sal89A]. The following is a brief description.

To construct an NSLR(1) parser, we precompute FIRST*, LAST*, FOLLOW and the three following sets.

$$\text{VISIBLE} = \{Y \mid Y \Rightarrow^* y, \text{ for some } y,\ y \neq \epsilon\}$$

$$\text{NEEDED\_FOLLOW}(A) = \{Y \mid \text{there is a rule } B \rightarrow \alpha X \beta Y \gamma,\ A \in \text{LAST}(X),\ \beta \Rightarrow^* \epsilon\}$$

$$\text{UNRESOLVABLE}(A) = \{Y \in \mathit{VISIBLE} \mid A \Rightarrow \alpha Y \beta,\ \alpha \Rightarrow^+ \epsilon\}$$

The function VISIBLE represents the set of nonterminals that can generate a non-empty word.

The set NEEDED_FOLLOW(A) needs a detailed description. If the definition is restated without the β intervening between X and Y, it is exactly the definition given by Tai for LM_FOLLOW, which contains the symbols that can immediately follow A in any sentential form of a leftmost derivation from the start symbol. NEEDED_FOLLOW(A) is different in that it contains those symbols that can follow A in a leftmost derivation even if they are separated by a string that can generate ϵ. The set NEEDED_FOLLOW(A) is so named because it contains those lookahead symbols that must not be noncanonically reduced to resolve a conflict involving a reduction to A. A symbol Y in NEEDED_FOLLOW(A) may not be reduced because there is a grammar rule involving A or symbols in $\text{LAST}^{-1}(A)$

that can only be used during a derivation if lookahead symbol Y remains unreduced. FOLLOW(A) is a closure of NEEDED_FOLLOW(A)—the union of the FIRST sets of symbols in NEEDED_FOLLOW(A).

Loop until all states completed.
  Compute next SLR state s_q for grammar.
  To each complete item $I_i = [A \rightarrow \alpha\bullet]$ attach
    a lookahead set L_i = FOLLOW(A)∩VISIBLE
    and a needed-lookahead set NL_i = NEEDED_FOLLOW(A)∩VISIBLE
  If state s_q has a conflict then
    For each conflicting lookahead symbol X:
      resolved := true
      For each complete item $[I_i = A \rightarrow \alpha\bullet]$ such that X is in L_i:
        If X is in NL_i or X is in UNRESOLVABLE(B) for some B in L_i
          then *resolved* := false
        else
          For each rule $B \rightarrow X\beta$ where B is in L_i:
            Add item $[B \rightarrow \bullet X\beta]$ to s_q
          end for
        end if
      end for
      If *resolved* then remove X from L_i
      else grammar is not NSLR(1).
      end if
    end for
  end if
end loop.

Figure 3.4: The construction algorithm for NSLR(1) parsers.

The list of UNRESOLVABLE lookahead nonterminals for a given nonterminal arises from the method of Cormack and Salomon for handling invisible nonterminals during conflict resolution. Its purpose is to reject grammars that require state expansion to occur

recursively on some states[1].

The NSLR(1) construction algorithm is given in Figure 3.4. After performing the state expansion algorithm, s_1 will have been expanded by adding noncanonical items, and the lookahead sets will have been trimmed. Figure 3.5 shows the NSLR(1) parser which accepts grammar G_1. Ignoring the question of *ϵ-productions* and invisible nonterminals, we are simply including nonterminals in the lookahead sets, and whenever a symbol Y causes a conflict in state s_q, we eliminate it by adding $[B \rightarrow \bullet Y\beta]$ for all productions such that B is in FIRST$^{-1}(Y)$. Therefore, for an NSLR(1) parser, there are three types of items in a state, kernel items, closure items, which exist in canonical parsers, and noncanonical items, which are added by the expansion algorithm, and can themselves generate closure items.

3.4 The Objective: Eliminating Useless Actions and States

One deficiency of parsers built by the existing NSLR(1) construction algorithm is that some transitions and states will never be accessed during parsing because of the transitions and states added for noncanonical parsing. For example, consider state s_{10} in Figure 3.5. The transition from s_5 to s_{10} is redundant because every c in the input string will be reduced to E in state s_1 before s_5 is entered. After the transition from s_5 to s_{10} is deleted, there will be no transitions to s_{10} and thus s_{10} will be redundant. Some lookahead symbols associated with reduce items could be useless too. For example, symbol A and B in state s_1 and symbol c in state s_{10} and s_{12} are useless. There are two kinds of useless lookahead symbols. One kind is *higher-level* nonterminals, which cannot have been reduced when the corresponding

[1]When the algorithm marks a conflict as unresolvable, it may still be possible to resolve the conflict for some grammars by adding ϵ-reducing items and more shift items. For details, please see [Sal89A]

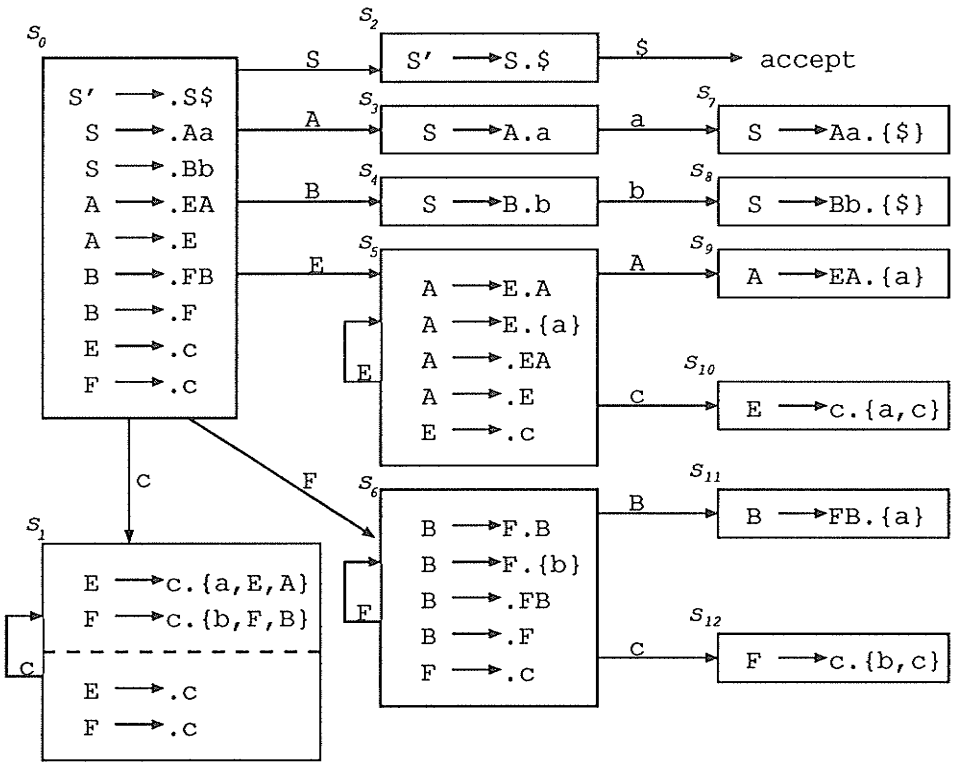

Figure 3.5: NSLR(1) parser for G_1 by expanding algorithm.

state is entered. Another kind is *lower-level* terminals and nonterminals, which have been reduced already and therefore cannot appear on the top of lookahead stack when a certain state is entered. Salomon shows how to delete most of the useless nonterminal symbols by using algorithm DUR2 [Sal89A]. In this thesis we will examine ways of removing redundant transition and states as well as removing more redundant lookahead symbols.

$$
\begin{array}{ll}
B \rightarrow B_1\beta_1 & A \rightarrow \alpha_1 A_1 \\
B_1 \rightarrow B_2\beta_2 & A_1 \rightarrow \alpha_2 A_2 \\
B_2 \rightarrow B_3\beta_3 & A_2 \rightarrow \alpha_3 A_3 \\
B_3 \rightarrow b\beta_4 & A_3 \rightarrow \alpha_4 a \\
C \rightarrow \theta A B \gamma &
\end{array}
$$

Figure 3.6: Part of the productions for grammar G_2.

In order to explain how the useless *shift* and *reduce* actions are generated, we give a more general example in the following discussion. Only part of the grammar and part of the parser are presented here. Assume that part of grammar G_2 is as in Figure 3.6:

Figure 3.7 shows part of the NSLR(1) parser for G_2 obtained by expanding the SLR(1) parser. (The item sets and lookahead sets shown there are also incomplete.) For that figure,

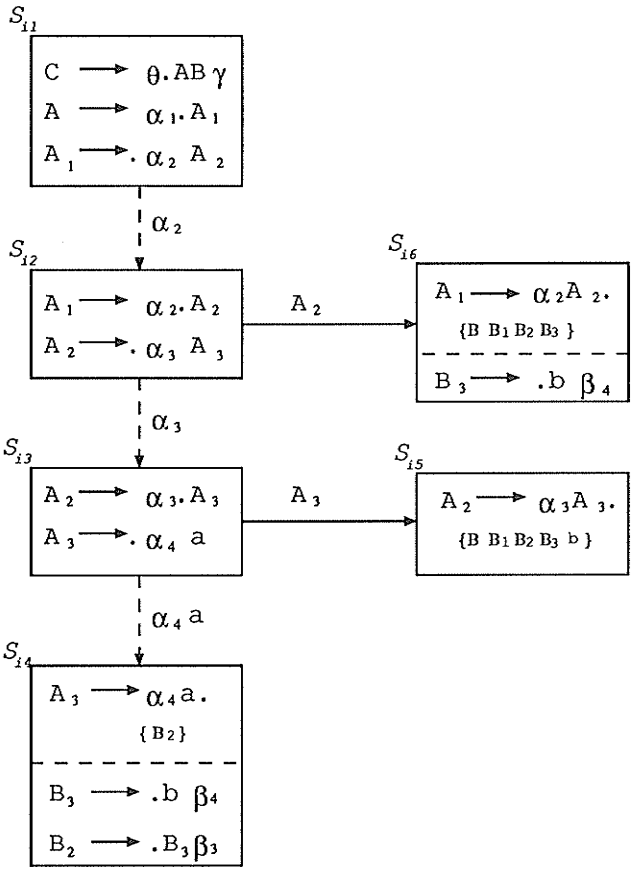

Figure 3.7: Part of the parser for grammar G_2.

$b \subseteq$ T-Follow(A_3), and $\{B, B_1, B_2, B_3, b\} \subseteq Follow(A_3)$.

Assume that for some reason, s_{i4} and s_{i6} are not SLR(1) consistent. In state s_{i4}, noncanonical expansion items $B_3 \rightarrow \bullet b\beta_4$ and $B_2 \rightarrow \bullet B_3\beta_3$ have been added. Item $B_3 \rightarrow \bullet b\beta_4$ is also added to state s_{i6} to resolve a conflict. The lookahead set of reduce item

$A_3 \rightarrow \alpha_4 a$ in state s_{i4} is $\{B_2\}$. After the phrase $B_3\beta_3$ is shifted in, and B_2 is pushed onto the lookahead stack by reducing the phrase $B_3\beta_3$, it will serve as the lookahead symbol for the reduce item $A_3 \rightarrow \alpha_4 a$. After phrase $\alpha_4 a$ has been reduced to A_3, s_{i4} will be popped from the state stack so that s_{i3} appears on the top of the state stack, and the top of the lookahead stack will be $A_3 B_2$. After state s_{i3} *shifts* A_3 and enters state s_{i5}, B_2 will be on top of the lookahead stack. In this case, B_3 and b are *lower-level* useless lookahead symbols since B_3 and b have been shifted and reduced to *higher-level* symbols (B_2, in this example) before state s_{i5} is entered. B and B_1 are *higher-level* useless lookahead symbols since item $A_2 \rightarrow \alpha_3 A_3$ is reduced before $B_1\beta_1$ is reduced to B, and $B_2\beta_2$ is reduced to B_1.

It is interested to notice that noncanonical items can be useless items too. In this example, $B_3 \rightarrow \bullet b\beta_4$ in state s_{i6} is a useless noncanonical item because of the state expansion of s_{i4}.

Chapter 4

The Properties of NSLR(1) Automata

The purpose of this chapter is to explore the essential properties of noncanonical SLR(1) automata. We are interested in the properties that such a parser would have if it contained no useless shift or reduce actions caused by noncanonical state expansion. By analyzing these properties we hope to discover methods for deleting useless actions. In the following sections we will define and analyze relations and sets for such parsers.

4.1 The Difference Between The Two Definitions of Reduce Actions

As mentioned in Chapter 3, we use a nontraditional definition of the reduce action. In the traditional definition [Aho86], the popping of a handle and the pushing of the nonterminal resulting from the reduction onto the state stack are performed in one action, so that

nonterminals never appear on the lookahead stack. The contents of a lookahead stack during a parser are a string in V_T^*. For example, assume state s_s corresponds to the item set in Figure 4.1, only terminal symbols, such as b, can be on the top of the lookahead stack when s_s is on the top of the state stack. The symbols B_1, B_2, B_3, and B_4 will all be shifted immediately after their reduction, and hence will not be pushed onto the lookahead stack.

$$
\begin{array}{|l|}
\hline
A \rightarrow C \bullet B_1 \\
B_1 \rightarrow \bullet B_2 \beta_2 \\
B_2 \rightarrow \bullet B_3 \beta_3 \\
B_3 \rightarrow \bullet B_4 \beta_4 \\
B_4 \rightarrow \bullet b \beta_5 \\
\hline
\end{array}
$$

Figure 4.1: State s_s.

In the new definition of *reduce*, the pop and the push are separated into two actions. A reduction on item $I = [A \rightarrow \alpha\bullet]$ consists of popping $|\alpha|$ states off of the state stack, and pushing the symbol A onto the lookahead stack. The next step depends on the state stack and the lookahead stack. Under this definition, any of the symbols B_1, B_2, B_3, B_4, and b can appear on the lookahead stack when the current state is s_s.

4.2 The shift and reduce Lookahead Sets

In order to explain parser properties clearly, we divide symbols on the lookahead stack into two categories, ShiftLA, and ReduceLA. ShiftLA(s_p) is a symbol set associated with all shift actions out of state s_p; ReduceLA(s_p, I) is a symbol set associated with reductions due to item I in state s_p.

The formal definition of the shift lookahead set for a state s_p is:

Definition 4.1

$$\begin{aligned}\text{ShiftLA}(s_p) \;=\; \{&Y \in V \mid [\alpha]Y\gamma \\ &\vdash_{shift} [\alpha Y]\gamma \\ &\vdash^* [S'\$], \text{ and } \alpha \text{ accesses } s_p\}\end{aligned}$$

Notice that both nonterminals and terminals are possible in shift lookahead sets. For canonical LR(1) parsers with the new definition of reduction, shift lookahead sets are easy to calculate. Knowing kernel items, we can get closure items using FIRST* sets. For example, if $X \in \text{ShiftLA}(s_p)$, and $Y \in \text{FIRST}^*(X)$, then it must be true that $Y \in \text{ShiftLA}(s_p)$. For a noncanonical LR(1) parsers, things are more complicated. The fact that $X \in \text{ShiftLA}(s_p)$ does not guarantee that $Y \in \text{FIRST}^*(X)$ will be useful in the shift lookahead set. It is possible that a string starting with Y will always be reduced to X by a noncanonical expansion item before state s_p is reached. In this case, Y should not be included in $\text{ShiftLA}(s_p)$.

Let the reduce lookahead set for item $[A \rightarrow \omega\bullet]$ in s_p be denoted by $\text{ReduceLA}(s_p, [A \rightarrow \omega\bullet])$,

Definition 4.2

$$\begin{aligned}\text{ReduceLA}(s_p, [A \rightarrow \omega\bullet]) \;=\; \{&Y \in V \mid [\alpha\omega]Y\gamma \\ &\vdash_{[A\rightarrow\omega\bullet]} [\alpha]AY\gamma \\ &\vdash^* [S'\$], \textit{and } \alpha\omega \textit{ accesses } s_p\}.\end{aligned}$$

Notice that αA need not be a viable prefix of a noncanonical sentential form. It is possible

for the string α to be reduced to another string when the top of the lookahead stack is A.

The symbols in ReduceLA(s_p, $I = [A \rightarrow \omega\bullet]$) can be terminals which were in the input string, as well as nonterminals which were pushed by previous *reduce* actions. For any noncanonical SLR(1) consistent state s_p, ReduceLA(s_p, I) $\cap$ ReduceLA(s_p, J) $= \emptyset$, and, $ReduceLA(s_p, I) \cap \text{ShiftLA}(s_p) = \emptyset$, for any items I and J in state s_p, where $I \neq J$.

ReduceLA(s_p) is a short form used to represent all symbols associated with reductions in state s_p.

Definition 4.3 $\text{ReduceLA}(s_p) = \bigcup_I \text{ReduceLA}(s_p, I)$, *where I is a complete item in state s_p.*

4.3 The Functions AccessLA and PreShifted

We define AccessLA(s_p) as the set of symbols which may appear on top of the lookahead stack when state s_p is entered from other states by shifting A, the accessing symbol of s_p. We will show later that actions in a state are determined by its kernel items and its AccessLA set. More precisely, AccessLA can be defined as:

Definition 4.4

$$\begin{aligned}\text{AccessLA}(s_p) \;=\; & \{Y \in V \mid [\alpha]AY\gamma \\ & \vdash_{shift} [\alpha A]Y\gamma \\ & \vdash^* [S'\$], \alpha A \text{ accesses } s_p, \\ & \alpha \text{ accesses some state } s_q, \textit{ and } s_q \neq s_p\}.\end{aligned}$$

The above definition deserves further explanation.

First, for canonical LR parsers, only terminals can be on the lookahead stack, so AccessLA(s_p) $\subseteq$ V_T; but for noncanonical LR parsers, AccessLA(s_p) $\subseteq$ V.

Second, symbols that can be on top of the lookahead stack only when state s_p is uncovered on the top of the state stack by reduce actions are not included in AccessLA(s_p).

Third, condition $s_q \neq s_p$ in Definition 4.4 is crucial. The existence of at least one such state s_q from which state s_p can be entered is obvious. For those states which can be entered by shifting from itself, this condition rules out some symbols which can be on top of the lookahead stack only when the state is entered from itself.

Please notice the differences between AccessLA(s_p), ReduceLA(s_p, I) and ShiftLA(s_p). They all contain symbols which can appear on top of the lookahead stack when the parser is in state s_p. When state s_p is entered from another state by shifting the accessing symbol A, any symbol Y that can be on top of the lookahead stack belongs to AccessLA(s_p). After more symbols are shifted, this string beginning with Y can be reduced to another symbol C. Symbol C may belong to ReduceLA(s_p, I) for some complete item I, or ShiftLA(s_p). But C is not in AccessLA(s_p), even if after shifting C, another instance of state s_p is pushed onto the state stack (in this case, symbol C is the same symbol as the accessing symbol A).

$$
\begin{array}{|l|}
\hline
A \rightarrow E \bullet A \\
A \rightarrow E \bullet \{a\} \\
A \rightarrow \bullet EA \\
A \rightarrow \bullet E \\
\hline
\end{array}
$$

Figure 4.2: State s_5 of the NSLR(1) parser for G_1.

We choose the NSLR(1) parser for G_1 as an example. State s_5 appears in Figure 4.2 after the useless item $E \rightarrow \bullet c$ is removed. (See also Figure 3.5).

In this parser, state s_0 is in PRED(s_5). When E is shifted by state s_0, and state s_5 is pushed onto the state stack, the terminal symbol a, or nonterminal symbol E may appear on top of the lookahead stack depending on the input sentence. Symbols a and E are in AccessLA(s_5). Symbol A may appear on top the of lookahead stack when the current state is s_5, but A is not in AccessLA(s_5), since there are no noncononical reductions to A that could leave A on the lookahead state without an immediate shift.

Property 4.1 *For any state* s_p, AccessLA(s_p) $\subseteq$ ShiftLA(s_p) $\cup$ ReduceLA(s_p).

Proof: This is an immediate consequence of the definitions of AccessLA, ShiftLA, and ReduceLA, since every symbol in AccessLA must lead to either a shift action or a reduce action. □

Property 4.2 *For any* $Y \in$ ShiftLA(s_p) $\cup$ ReduceLA(s_p), *there exists an* X, *such that* $X \in$ AccessLA(s_p) *and* $X \in$ FIRST*(Y).

Proof: For any $Y \in$ ShiftLA(s_p) $\cup$ ReduceLA(s_p), there are only two possible cases:

1. there is no item with Y as the left-part in state s_p, or
2. there is at least one item with Y as the left-part.

Case 1: In this case, Y must be in AccessLA(s_p). By way of obtaining a contradiction, suppose that $Y \notin$ AccessLA(s_p). According to the definition of AccessLA, there exists

an α and an A such that

$$[\alpha]A\gamma \vdash_{shift} [\alpha A]\gamma \vdash^* [S'\$]$$

where α accesses s_q, αA accesses s_p, and $s_p \neq s_q$. Because $Y \notin \text{AccessLA}(s_p)$, Y cannot be on the top of the lookahead state when s_p is entered by shifting A from state s_q. Since $Y \in \text{ShiftLA}(s_p) \cup \text{ReduceLA}(s_p)$, Y can be on the top of the lookahead stack when the current state is s_p. The only way that Y can be pushed onto the lookahead stack is by shifting a string from state s_p, and then reducing this string to Y. The first symbol X of this string must belong to set $\text{ShiftLA}(s_p) \cup \text{ReduceLA}(s_p)$ and $Y \Rightarrow^+ X\theta$. So, we conclude that in state s_p, there must be at least one item whose left part is Y. This contradicts the assumption. So that $Y \in AccessLA(s_p)$, and the property is true in case 1.

Case 2: Since there is at least one item with Y as its left-part, this item can be written as $[Y \rightarrow \bullet Y_1\theta_1]$, furthermore, we can find all items of the form $[Y_1 \rightarrow \bullet Y_2\theta_2]$, $[Y_2 \rightarrow \bullet Y_3\theta_3]$, ..., $[Y_n \rightarrow \bullet X\theta_{n+1}]$ that are in state s_p, and X is not the left part of any item in this state. According to the proof of case 1, $X \in \text{AccessLA}(s_p)$, so that the property is true in case 2.

Hence we proved property 4.2. □

Property 4.2 tells us that AccessLA can be used to limit which symbols can be shift or reduce lookahead symbols.

Property 4.3 *AccessLA(s_p)$\subseteq V_T$, if s_p is accessed by a terminal symbol a.*

Proof: The lookahead stack is initialized with all the input terminal symbols. During parsing, only nonterminals can be pushed onto the lookahead stack by reduce actions. Due to the nature of stacks, the contents of a lookahead stack (from top to bottom) must be a string in $V_N^* V_T^*$. This means that only terminals can be under a terminal in the lookahead stack. Thus the symbol exposed on the lookahead stack must be a terminal, if state s_p is accessed by shifting a terminal symbol a. □

Having defined AccessLA, we now define the function PreShifted. The function AccessLA defines a set of lowest-level symbols which can appear on the lookahead stack for a state. The function PreShifted for a state s_p is defined as a set of symbols which are always shifted by noncanonical expansion items before state s_p is entered, and therefore these symbols cannot be shift or reduce lookahead symbols in this state. PreShifted(s_p) can be computed from AccessLA sets and FIRST sets.

Definition 4.5

$$\begin{aligned} \text{PreShifted}(s_p) \;=\; & \{Y \mid \exists A \in \text{AccessLA}(s_p),\ Y \in \text{FIRST}^+(A), \\ & \text{for any } B,\ B \in \text{AccessLA}(s_p),\ Y \notin \text{FIRST}^{-1}(B).\} \end{aligned}$$

From the definition, we can see that the symbols in PreShifted(s_p) are lower-level symbols of those in AccessLA(s_p).

Property 4.4 $\text{PreShifted}(s_p) \cap (\text{ShiftLA}(s_p) \cup \text{ReduceLA}(s_p)) = \emptyset$

Proof: We prove this property by contradiction: suppose the property does not hold, then there is some Y,

$$Y \in \text{PreShifted}(s_p) \tag{4.1}$$

$$Y \in \text{ShiftLA}(s_p) \cup \text{ReduceLA}(s_p) \tag{4.2}$$

Relation 4.2 with Property 4.2 imply that there exists an X, such that $X \in \text{AccessLA}(s_p)$ and $X \in \text{FIRST}^*(Y)$. By Definition 4.5, $Y \notin \text{PreShifted}(s_p)$. Hence there is a contradition with the assumption, and we have proved the property. □

Property 4.5 *PreShifted(s_p)= $\emptyset$, if s_p is accessed by a terminal symbol a.*

Proof: This proof is trivial. By Property 4.3, if s_p is accessed by terminal symbol a, $\text{AccessLA}(s_p) \subseteq V_T$. This implies that there is no A, such that $Y \in \text{FIRST}^+(A)$, so that $\text{PreShifted}(s_p) = \emptyset$ by Definition 4.5. □

4.4 The LOOKBACK Function

Here we define another function called LOOKBACK, which is very similar to the relation *Lookback* defined by DeRemer [DeRe82].

Intuitively, we say s_q is in $\text{LOOKBACK}(s_p, I = [A \rightarrow \alpha\bullet])$ if s_q is the state which pushes a nonterminal symbol A onto the lookahead stack by reduction of item I, and s_p is the state which is entered by shifting this A. For example, in Figure 3.7, s_{i4} is in

LOOKBACK($s_{i5}, [A_3 \rightarrow \alpha_4 a \bullet]$) and s_{i5} in LOOKBACK($s_{i6}, [A_2 \rightarrow \alpha_3 A_3 \bullet]$). We simply say s_q *lookback* s_p if no confusion arises.

Before we formally define the LOOKBACK function, we first introduce two relations related to that function.

Definition 4.6 *State s_q UIRS (uninterrupted reduce and shift to) state s_p at item $I = [A \rightarrow \omega \bullet]$ if, there is at least one state s_s, $s_s \xrightarrow{\omega} s_q$, $s_s \xrightarrow{A} s_p$. and for any $Y \in$ ReduceLA(s_q, I), there is a path,*

$$Top([s_s : \omega])\, Y\gamma \vdash_{[A \rightarrow \omega \bullet]} Top([s_s])\, AY\gamma$$
$$\vdash_{shift} Top([s_s : A])\, Y\gamma.$$

Figure 4.3 shows part of a parser for an NSLR(1) grammar in which state s_q UIRS state s_p at item $[A \rightarrow \alpha \bullet]$.

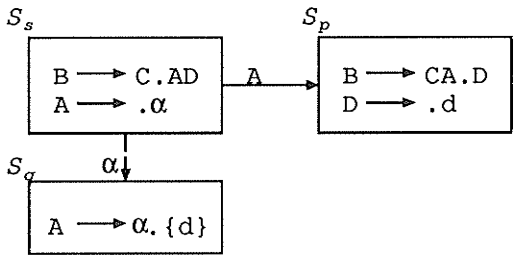

Figure 4.3: State s_q UIRS state s_p.

Definition 4.7 *State s_q IRS (interrupted reduce and shift to) state s_p at item $I = [A \rightarrow \omega \bullet]$ if there is a state s_s, and $B \Rightarrow^+ \beta$, such that $s_s \xrightarrow{\beta\omega} s_q$, and $s_s \xrightarrow{BA} s_p$, for any $Y \in ReduceLA(s_q, I)$, there is a path,*

$$Top([s_s : \beta\omega])\, Y\gamma \vdash_{A \rightarrow \omega} Top([s_s : \beta])\, AY\gamma$$
$$\vdash_{shift} Top([s_s : BA])\, Y\gamma$$

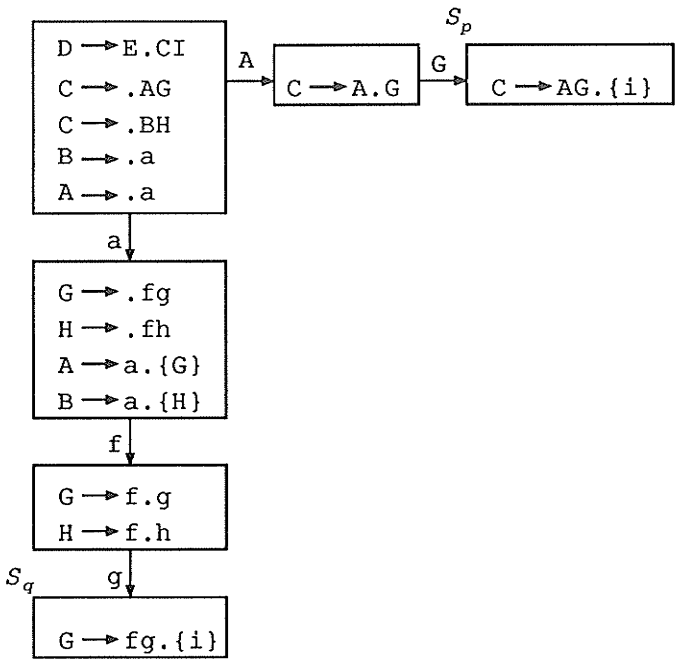

Figure 4.4: State s_q IRS state s_p.

Figure 4.4 gives an example in which state s_q IRS state s_p at item $[G \rightarrow fg\bullet]$.

The IRS relation is unique for noncanonical automata. For a parser which is canonical LR consistent, only UIRS relations hold for states.

Formally, function LOOKBACK is defined as follows:

Definition 4.8 *Function* LOOKBACK, *for a state s_p and an item $I = [A \rightarrow \omega\bullet]$, is defined as a set of states. State s_q is in* LOOKBACK *of s_p at item I, if state s_q UIRS state s_p at item I, or state s_q IRS state s_p at item I, we write as* $s_q \in \text{LOOKBACK}(s_p, I = [A \rightarrow \omega\bullet])$.

Observing the relationship of the symbols in AccessLA(s_p) and ReduceLA(s_p), we obtain Theorem 4.1, which helps us to determine the symbols in AccessLA for a given action table.

Theorem 4.1

$$\text{AccessLA}(s_p) \subseteq \bigcup_{s_q} \text{ReduceLA}(s_q, [A \rightarrow \omega\bullet]),$$

where $s_q \in \text{LOOKBACK}(s_p, [A \rightarrow \omega\bullet])$.

Proof: According to the definition of AccessLA, for any $Y \in \text{AccessLA}(s_p)$, and for some θ and γ, there is a transition $[\theta]AY\gamma \vdash_{shift} [\theta A]Y\gamma$, such that θ accesses s_w, θA accesses s_p, and $s_w \neq s_p$.

Since A is a nonterminal, there must be some strings α and ω, and a state s_q accessed by $\alpha\omega$, in which a string ω is reduced to A, such that

$$[\alpha\omega]Y\gamma \vdash_{A\rightarrow\omega} [\alpha]AY\gamma \vdash^* [\theta]AY\gamma \vdash_{shift} [\theta A]Y\gamma.$$

If there were no such α and ω, then there would be no deriation $S' \rightarrow^+ [\theta]AY\gamma$.

Here two points need to be explained:

1. $Y \in \text{ReduceLA}(s_q, I = [A \rightarrow \omega])$ simply because Y is under A on the lookahead stack while the parser is in state s_w, and A is reduced from string ω. The A must have been pushed by a reduce action and Y must have been the lookahead symbol for that action.

2. $Top([\alpha])$ may differ from $Top([\theta])$. After the ω is reduced to A, a sequence of reduce and shift moves may occur while this A remains on the lookahead stack.

Now, we want to prove that $s_q \in \text{LOOKBACK}(s_p, I = [A \rightarrow \omega\bullet])$.

For any $X \in \text{ReduceLA}(s_q, I)$, there are some γ', such that $[\alpha\omega]X\gamma' \vdash_{A\rightarrow\omega} [\alpha]AX\gamma'$.

Since $[\alpha]AY\gamma \vdash^* [\theta]AY\gamma$, A remains on the lookahead stack, and $s_w \xrightarrow{A} s_p$, so

$$[\alpha\omega]X\gamma' \vdash_{A\to\omega} [\alpha]AX\gamma' \vdash^* [\theta]AX\gamma' \vdash_{shift} [\theta A]X\gamma'$$

We conclude that $s_q \in \text{LOOKBACK}(s_p, I = [A \to \omega\bullet])$.

Hence, we proved that $\text{AccessLA}(s_p) \subseteq \cup\text{ReduceLA}(s_q, [A \to \omega\bullet])$, where $s_q \in \text{LOOKBACK}(s_p, [A \to \omega\bullet])$. □

In this chapter, we defined sets and analyzed relations between them. We showed that $PreShifted(s_p)$ is a set of symbols which cannot be in this state s_p. In next chapter, I will present algorithms to detect subsets of PreShifted, given an action table containing useless actions.

Chapter 5

Algorithms for Shrinking NSLR(1) Parsers

In chapter 4, we analyzed the properties of the NSLR(1) parsers without useless actions caused by noncanonical state expansion. In fact, up to now there does not exist a practical parser construction technique which can generate such parsers. Given action tables generated by an existing NSLR(1) parser construction algorithm NPG_{SC} (as presented by Salomon and Cormack [Sal89B]), we present two approaches for deleting useless actions by using the relations we found in Chapter 4. The safety of these approaches is also proved in this chapter. The results of the two approaches are analyzed in Section 5.4. The results for real character-level grammars for programming languages are poor. The reasons for the poor performance are analyzed and the analysis leads to a simple improvement of the parser construction algorithm. When applied to parsers generated by the modified parser construction algorithm, the algorithms for deleting useless parser actions lead to significantly smaller

parsers.

5.1 Lookahead Sets Obtained from Action Tables

The algorithms presented here are designed to work on the parser action tables built by the standard LR(1) family of parser construction algorithms. In order to differentiate sets which are obtained from an actual action table from theoretical sets which contain no useless actions, we precede the names of the former sets with the prefix *AT*. ATReduceLA and ATShiftLA, are respectively the *reduce* and *shift* lookahead sets computed from an action table.

$$\text{ATShiftLA}(s_p) = \{Y \mid Action(s_p, Y) = shift\} \tag{5.1}$$

$$\text{ATReduceLA}(s_p, I = [A \rightarrow \omega\bullet]) = \{Y \mid Action(s_p, Y) = reduce\ by\ A \rightarrow \omega\} \tag{5.2}$$

ATReduceLA(s_p) is defined as the set of symbols associated with any reduction in state s_p.

$$\text{ATReduceLA}(s_p) = \{Y \mid Action(s_p, Y) = reduce\} \tag{5.3}$$

For any NSLR(1) consistent state s_p, ATReduceLA(s_p, I) $\cap$ ATReduceLA(s_p, J) $= \emptyset$, and, ATReduceLA(s_p, I) $\cap$ ATShiftLA(s_p) $= \emptyset$, for any items I and J in state s_p, where $I \neq J$.

The sets of actions obtained from an action table are supersets of the corresponding theoretically optimal ones.

$$\text{ATShiftLA}(s_p) \supseteq \text{ShiftLA}(s_p) \tag{5.4}$$

$$\text{ATReduceLA}(s_p, [A \rightarrow \omega\bullet]) \supseteq \text{ReduceLA}(s_p, [A \rightarrow \omega\bullet]) \tag{5.5}$$

$$\text{ATReduceLA}(s_p) \supseteq \text{ReduceLA}(s_p) \tag{5.6}$$

5.2 A Simple Parser Shrinking Algorithm

The algorithm presented in this section is simple and quick because useless actions in a parser are detected by examining each state of the parser individually without considering the paths between states.

The principle of this algorithm is that if some symbol $X \in \text{FOLLOW}(A)$ is deleted by noncanonical state expansion from the lookahead set of all complete items $[A \rightarrow \omega\bullet]$ for all ω in all states, then X cannot possibly be on the lookahead stack after shifting A. In other words, X should not be in $\text{ATAccessLA}(A)$.

5.2.1 ATAccessLA and ATPreShifted

ATAccessLA and ATPreShifted for a nonterminal symbol A can be computed as:

$$\text{ATAccessLA}(A) = \bigcup_{s_p \in K, \omega} \text{ATReduceLA}(s_p, [A \rightarrow \omega\bullet]) \tag{5.7}$$

$$\begin{aligned} \text{ATPreShifted}(A) \;=\; & \{Y \mid \exists\, C \in \text{ATAccessLA}(A), \\ & Y \in \text{FIRST}^{+}(C), \text{ and for any} \\ & B \in \text{ATAccessLA}(A),\ Y \notin \text{FIRST}^{-1}(B)\} \end{aligned} \tag{5.8}$$

We will show that useless actions can be detected with the set ATPreShifted. First, we want to prove that:

$$\text{ATAccessLA}(A) \supseteq \text{AccessLA}(s_p) \tag{5.9}$$

where A is the accessing symbol of state s_p.

Proof: By the definition of ATAccessLA in Relation 5.7,

$$\text{ATAccessLA}(A) = \bigcup_{s_q \in K} \text{ATReduceLA}(s_q, [A \rightarrow \omega\bullet]).$$

By Relation 5.5, we know that $\text{ReduceLA}(s_q, [A \rightarrow \omega\bullet]) \subseteq \text{ATReduceLA}(s_q, [A \rightarrow \omega\bullet])$, so that

$$\text{ATAccessLA}(A) \supseteq \bigcup_{s_q \in K} \text{ReduceLA}(s_q, [A \rightarrow \omega\bullet]).$$

If we can prove that

$$\bigcup_{s_q \in K} \text{ReduceLA}(s_q, [A \rightarrow \omega\bullet]) \supseteq \text{AccessLA}(s_p),$$

then the relation is proved.

According to Theorem 4.1,

$$\bigcup_{s_q} \text{ReduceLA}(s_q, [A \rightarrow \omega\bullet]) \supseteq \text{AccessLA}(s_p),$$

where $s_q \in \text{LOOKBACK}(s_p, [A \rightarrow \omega\bullet])$.

By enlarging the leftside, we have

$$\bigcup_{s_q \in K} \text{ReduceLA}(s_q, [A \rightarrow \omega\bullet]) \supseteq \text{AccessLA}(s_p),$$

and so that the relation 5.9 is proved. □

Based on Relation 5.9, we can prove: if A is the accessing symbol of state s_p, then actions on the lookahead symbols in $\text{ATPreShifted}(A) \cap (\text{ATShiftLA}(s_p) \cup \text{ATReduceLA}(s_p))$ form a subset of useless actions in state s_p.

This statement can be explained in another way: for a state s_p, which is accessed by A,

$$\begin{aligned} & \text{ATPreShifted}(A) \cap (\text{ATShiftLA}(s_p) \cup \text{ATReduceLA}(s_p)) \\ \subseteq \quad & (\text{ATShiftLA}(s_p) \cup \text{ATReduceLA}(s_p)) - (\text{ShiftLA}(s_p) \cup \text{ReduceLA}(s_p)) \end{aligned} \tag{5.10}$$

Proof: We first let (i) $Y \in \text{ATPreShifted}(A) \cap (\text{ATShiftLA}(s_p) \cup \text{ATReduceLA}(s_p))$, assume (ii) $Y \notin (\text{ATShiftLA}(s_p) \cup \text{ATReduceLA}(s_p)) - (\text{ShiftLA}(s_p) \cup \text{ReduceLA}(s_p))$, and prove that (i) and (ii) are inconsistent.

Based on hypothesis (i), we have $Y \in \text{ATPreShifted}(A)$ and $Y \in \text{ATShiftLA}(s_p) \cup \text{ATReduceLA}(s_p)$. Then $Y \in \text{ShiftLA}(s_p) \cup \text{ReduceLA}(s_p)$ must be true by hypothesis (ii). With Property 4.2, we know that there exists an X, such that $X \in \text{AccessLA}(s_p)$ and $X \in \text{FIRST}^*(Y)$ or we say $Y \in \text{FIRST}^{-1}(X)$. $X \in \text{ATAccessLA}(s_p)$ because of Relation 5.9. Then by the definition of $\text{ATPreShifted}(A)$ in 5.8, we conclude that $Y \notin \text{ATPreShifted}(A)$, which is a contradiction with our hypothesis (i).

Thus we proved that Relation 5.10 is true. □

From Relation 5.10, we conclude that given an action table, we can delete some useless actions and get a valid suboptimal parser by computing ATPreShifted sets for every nonterminal.

5.2.2 Algorithm DULA1:

Figure 5.1 presents a simple algorithm for deleting useless lookahead symbols from an action table.

This algorithm runs in time linearly proportional to the size of V_N. The safety of this algorithm can be ensured by the discussion in section 5.2.1. Useless actions of some parsing tables generated by *SOAP*, were identified using this algorithm. Table 5.1 in section 5.4 shows the results for grammars for PASCAL and Modula 2, and some grammars supplied by Kuo-Chung Tai [Tai79], named KCT1, KCT2, KCT3, and KCT4[1].

The algorithm is not powerful enough to delete all useless actions. Suppose that the same complete item $A \rightarrow \alpha\bullet$ appears in two different states s_p and s_q, and that in state s_p there is a conflict on a lookahead symbol Y, and in s_q there is no such conflict. Since there is no conflict in s_q, Y will remain in ATAccessLA(A) and actions on Y will remain in all states accessed by A whether or not these actions are useless. For example, in Figure 3.5, symbol c is include in ATAccessLA(A), since Action(s_{10}, c)=reduce, despite the fact that s_{10} is an unreachable state. This prevents Action(s_5, c) from being deleted. The second algorithm is proposed to improve performance.

[1]See Appendix A

/* Compute ATAccessLA(A) for every nonterminal A.*/
Initialize ATAccessLA(A) as the empty set for all A.
For every state s_s:
    For every X such that Action(s_s, X)= reduce by $A \rightarrow \alpha$:
        Include X in ATAccessLA(A)
    End for.
End for.

/* Compute ATPreShifted(A) for every nonterminal A. */
For every nonterminal A:
    Initialize sets High and Low to be empty.
    For every symbol X in ATAccessLA(A):
        High= High$\cup\{\text{FIRST}^{-1}(X)\}$
        Low= Low$\cup\{\text{FIRST}^{+}(X)\}$
    End for.
    ATPreShifted(A) = Low$-$ High
End for.

/* Delete useless actions. */
For each state s_s:
    A = accessing symbol of s_s.
    For each symbol Y in ATPreShifted(A).
        set action[s_s, Y]= error
    End for.
End for.

Figure 5.1: Algorithm DULA1: A simple algorithm to delete useless actions.

5.3 A Non-simple Algorithm

Given a state s_p, we can delete useless actions by retrieving all states s_q, which *lookback* s_p. If s_q UIRS s_p, it is easy to find s_q if we know s_p. But the problem of how to determine s_q for a given s_p, if s_q IRS s_p, is not as easy to solve. Figure 5.2 shows one example of a state $s_{q_{i+1}}$ that IRS another state $s_{p_{i+1}}$.

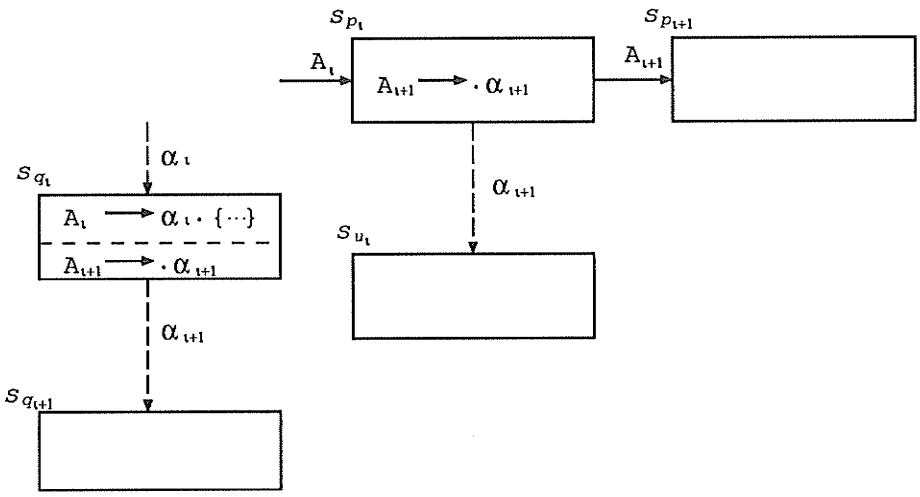

Figure 5.2: State $s_{q_{i+1}}$ lookback $s_{p_{i+1}}$.

Since there are some states like s_{u_i}, which seem to *lookback* $s_{p_{i+1}}$, but in fact, may be unreachable, we may have to look several steps backward and forward to find $s_{q_{i+1}}$. Here we provide a suboptimal algorithm that looks one step back. Instead of finding $s_{q_{i+1}}$, we are trying to compute ATAccessLA($s_{p_{i+1}}$) according to s_{u_i}, which is easy to compute.

5.3.1 ATAccessLA and ATPreShifted

ATAccessLA is designed to look one step back.

$$\text{ATAccessLA}(s_p) = \bigcup_{s_q, I} \text{ATReduceLA}(s_q, I = [A \rightarrow \omega\bullet]), \text{ where}$$

$$\exists s_u,\ s_u \xrightarrow{A} s_p,\ \text{and}\ s_u \xrightarrow{\omega} s_q. \tag{5.11}$$

(For this algorithm ATAccessLA(s_p) is not necessarily $\supseteq$ AccessLA.)

The ATPreShifted for a state s_p can be computed as:

$$\begin{aligned} \text{ATPreShifted}(s_p) \;=\; & \{Y \mid \exists\ B \in \text{ATAccessLA}(s_p), \\ & Y \in \text{FIRST}^{+}(B),\ \text{and for any} \\ & C \in \text{ATAccessLA}(s_p),\ Y \notin \text{FIRST}^{-1}(C)\}. \end{aligned} \tag{5.12}$$

Our algorithm is based on the fact that ATPreShifted(s_p)$\cap$(ATShiftLA(s_p)$\cup$ATReduceLA(s_p)) is a subset of useless actions in state s_p.

This statement is equivalent to the following Relation 5.13:

$$\begin{aligned} & \text{ATPreShifted}(s_p) \cap (\text{ATShiftLA}(s_p) \cup \text{ATReduceLA}(s_p)) \\ \subseteq\ & (\text{ATShiftLA}(s_p) \cup \text{ATReduceLA}(s_p)) - (\text{ShiftLA}(s_p) \cup \text{ReduceLA}(s_p)) \end{aligned} \tag{5.13}$$

In order to prove the above statement, we first prove:

Property 5.1 *Given an NSLR(1) grammar and the action table of that grammar obtained by performing the state expansion algorithm, if* $s_{q_i} \in \text{LOOKBACK}(s_{p_i}, [A_i \rightarrow \alpha_i \bullet])$, $[A_{i+1} \rightarrow \bullet\alpha_{i+1}]$ *is a noncanonical item in state* s_{q_i}, $s_{q_i} \xrightarrow{\alpha_{i+1}} s_{q_{i+1}}$, *and* $s_{p_i} \xrightarrow{\alpha_{i+1}} s_{u_i}$, *then the item set of* s_{u_i} *is smaller than that of* $s_{q_{i+1}}$.

Figure 5.2 may help us understand the states and their relation described in Property 5.1.

Proof: The closure items and the noncanonical items are generated from kernel items, so, if we can prove that the kernel item set in s_{u_i} is smaller than that of $s_{q_{i+1}}$, then the property is true.

Let $I = [B_{i+1} \rightarrow \theta_{i+1} \bullet \gamma_{i+1}]$ be an item in state s_{u_i}, we are trying to prove that I must be in $s_{q_{i+1}}$. There are three cases that must be considered.

Case 1. $\theta_{i+1} = \alpha_{i+1}$.

Item I in state s_{u_i} can be written as $I = [B_{i+1} \rightarrow \alpha_{i+1} \bullet \gamma_{i+1}]$ and $[B_{i+1} \rightarrow \bullet\alpha_{i+1}\gamma_{i+1}]$ is an initial item in state s_{p_i}. $B_{i+1} \in \text{FOLLOW}(A_i)$ because A_i is the accessing symbol of state s_{p_i}. With the given condition that $[A_{i+1} \rightarrow \bullet\alpha_{i+1}]$ is an noncanonical item in state s_{q_i}, we obtain that $[B_{i+1} \rightarrow \bullet\alpha_{i+1}\gamma_{i+1}]$ must be added to state s_{q_i} according to the state-expansion algorithm. So that $[B_{i+1} \rightarrow \alpha_{i+1} \bullet \gamma_{i+1}]$ is an item in state $s_{q_{i+1}}$ since $s_{q_i} \stackrel{\alpha_{i+1}}{\rightarrow} s_{q_{i+1}}$. So Property 5.1 holds for case 1.

Case 2. $\eta\alpha_{i+1} = \theta_{i+1}$ and $\eta \neq \epsilon$.

Item I in state s_{u_i} can be rewritten as $[B_{i+1} \rightarrow \eta\alpha_{i+1} \bullet \gamma_{i+1}]$. And $[B_{i+1} \rightarrow \eta \bullet \alpha_{i+1}\gamma_{i+1}]$ is an item in state s_{p_i}. The last symbol of η is A_i, since A_i is the accessing symbol of state s_{p_i}. Then the first symbol of α_{i+1} is in NEEDED_FOLLOW(A_i). According to the NSLR(1) construction algorithm NPG_{SC}, the grammar is not NSLR(1) consistent. So case 2 is impossible.

Case 3. $\eta_2\theta_{i+1} = \alpha_{i+1}$, $\eta_2 \neq \epsilon$.

Item $I = [B_{i+1} \rightarrow \theta_{i+1} \bullet \gamma_{i+1}]$ is in state s_{u_i}. There must be an item $[C \rightarrow \eta_1 \bullet \eta_2 B_{i+1}\zeta]$ in state s_{p_i}. There are two situations: $\eta_1 = \epsilon$, or not.

3.1. $\eta_1 = \epsilon$, $[C \rightarrow \bullet\eta_2 B_{i+1}\zeta]$ is in state s_{p_i}. $C \in \text{FOLLOW}(A_i)$ because A_i is the accessing symbol of state s_{p_i}. $[C \rightarrow \bullet\eta_2 B_{i+1}\zeta]$ will be in state s_{q_i}, since the first

symbol of α_{i+1} is the same as that of η_2. With the condition that $s_{q_i} \stackrel{\alpha_{i+1}}{\rightarrow} s_{q_{i+1}}$, we get that $[B_{i+1} \rightarrow \alpha_{i+1} \bullet \gamma_{i+1}]$ is an item in state $s_{q_{i+1}}$.

3.2. $\eta_1 \neq \epsilon$, $[C \rightarrow \eta_1 \bullet \eta_2 B_{i+1} \zeta]$ is in state s_{p_i}. The last symbol of η_1 is A_i. For the same reason as Case 2, the grammar is not NSLR(1) consistent.

Hence we prove in all cases that item I must be in state $s_{q_{i+1}}$, otherwise the grammar is not NSLR(1) consistent. □

From Property 5.1, we can conclude that:

Property 5.2 *For state s_{u_i} and $s_{q_{i+1}}$, which satisfy the conditions in Property 5.1, if $C \in$ ATReduceLA(s_{u_i}), then there is a $D \in$ ReduceLA($s_{q_{i+1}}$), and $C \in$ FIRST*(D)*

Proof: By the construction algorithm NPG_{SC}, we know that if $C \in$ ATReduceLA(s_{u_i}), there are items $[A \rightarrow \theta \bullet \{C\}]$ and $[C \rightarrow \bullet\gamma]$ in state s_{u_i}. By Property 5.1, both $[A \rightarrow \theta\bullet]$ and $[C \rightarrow \bullet\gamma]$ are in state $s_{q_{i+1}}$. The only possible way that $C \notin$ ReduceLA($s_{q_{i+1}}, [A \rightarrow \theta\bullet]$) is that there is some D, $D \in$ ReduceLA($s_{q_{i+1}}, [A \rightarrow \theta\bullet]$), and $[D \rightarrow \bullet C\eta]$ is an noncanonical item in state $s_{q_{i+1}}$. Hence, we proved the property. □

Now we are ready to prove the relation 5.13.

Proof: The proof here is similar to the proof of Relation 5.10 in Section 5.3. First we let (i) $Y \in$ ATPreShifted(s_p) $\cap$ (ATShiftLA(s_p) $\cup$ ATReduceLA(s_p)), assume (ii) $Y \notin$ (ATShiftLA(s_p) $\cup$ ATReduceLA(s_p)) $-$ (ShiftLA(s_p) $\cup$ ReduceLA(s_p)), and prove that assumption (ii) is a contradiction with assumption (i).

Since hypothesis (i) is true, therefore $Y \in$ ATPreShifted(s_p), and $Y \in$ ATShiftLA(s_p)$\cup$

ATReduceLA(s_p), With hypothesis (ii), the above results imply that $Y \in$ ReduceLA(s_p)$\cup$ ShiftLA(s_p). So that according to Property 4.1, there exists an X, such that $X \in$ AccessLA(s_p) and $X \in$ FIRST$^*(Y)$ or we say $Y \in$ FIRST$^{-1}(X)$. We know there is at least one path:

$$[\alpha\omega_i]X\gamma \vdash_{[A_i \to \omega_i \bullet]} [\alpha]A_iX\gamma \vdash^* [\theta]A_iX\gamma \vdash_{shift} [\theta A_i]X\gamma,$$

and θA_i accesses s_{p_i}, θ accesses $s_{p_{i-1}} \neq s_p$, and $\alpha\omega_i$ accesses s_{q_i}.

In the action table, there is a state s_{u_i}, $s_{p_{i-1}} \xrightarrow{\alpha_i} s_{u_i}$, and $[A_i \to \alpha_i \bullet]$ is an item in state s_{u_i}. We are going to prove that only X or $Z \in$ FIRST$^*(X)$ may belong to ATReduceLA(s_{u_i}, $[A_i \to \alpha_i \bullet]$).

For any $V \in$ FIRST$^{-1}(X)$, V can not be in set ATReduceLA(s_{u_i}). Otherwise, $[V \to \bullet X_1\eta_1]$, $[X_1 \to \bullet X_2\eta_2]$, $\cdots$, $[X_m \to \bullet X\eta_m]$, would be items in state s_{u_i}, and they are items in state $s_{q_{i-1}}$ too by the property 5.1. So that $X \notin$ ReduceLA($s_{q_{i-1}}$) since ShiftLA and ReduceLA are exclusive. It turns out that the path is impossible, which contradicts the condition that $X \in$ AccessLA(s_p).

Since there is an X or a $Z \in$ FIRST$^*(X)$ in ATReduceLA(s_{u_i}, $[A_i \to \alpha_i \bullet]$), with the definitions 5.11, and 5.12, we can conclude that $Y \notin$ ATPreShifted(s_p), which contradicts our hypothesis. Hence we proved that Relation 5.13 is true. □

The statement shows that we can delete some of the useless actions by computing ATAccessLA and ATPreShifted for every state and get a valid parser.

/* Compute ATAccessLA(s_p) for every state s_p.*/
Initialize ATAccessLA(s_p) as the empty set for every state s_p.
For every state s_p:
    Get the accessing symbol A of state s_p.
    Find all the state s_s, such that $s_s \xrightarrow{A} s_p$.
    For every state s_q, $s_q \xrightarrow{\alpha} s_s$, where $A \rightarrow \alpha \in P$:
        For every symbol X:
            If Action(s_q, X)= reduce from $[A \rightarrow \alpha\bullet]$, include X in ATAccessLA(s_p).
        End for.
    End for.
End for.

/* Compute ATPreShifted(s_p) for every state s_p */
For every state s_p:
    Initialize the sets High and Low to be empty.
    For every symbol X in ATAccessLA(s_p):
        High= High$\cup$ {FIRST$^{-1}(X)$ }
        Low= Low$\cup$ {FIRST$^{+}(X)$ }
    End for.
    ATPreShifted(A) = Low$-$ High
End for.

/* Delete useless actions */
For each state s_s:
    For each symbol Y in ATPreShifted(s_s).
        set action[s_s, Y]= error
    End for.
End for.

Figure 5.3: Algorithm DULA2: The second algorithm for deleting useless actions.

5.3.2 Algorithm DULA2

Figure 5.3 is an algorithm for the second method. The algorithm is linear. The safety of this algorithm can be ensured by the discussion in 5.3.1.

5.4 Results

Modules for performing the elimination of inaccessible parser actions were written in Modula-2 and added to the SOAP parser generator. A group of sample grammars were tested and verified manually. The optimal parse-table generated from PASCAL and Modula-2 grammars were verified on a test suite of programs that were supplied by Salomon. The remainder of this section discusses results gained from the implementation of the proposed techniques.

Grammar	KCT1	KCT2	KCT3	KCT4	PASCAL	MODULA-2
total no. of states	15	24	17	12	1005	1150
states added by non-canonical expansion	0	1	0	1	84	118
total no. of SHIFTs	17	26	22	12	14363	17217
total no. of REDUCEs	14	13	11	9	45026	52684
useless SHIFTs	2	4	9	1	?	?
useless REDUCEs	4	0	1	1	?	?
redundant states	2	2	4	1	?	?

Table 5.1: Useless actions in action tables.

Table 5.1 summarizes the useless actions and redundant states in the action tables of a group of grammars[2]. Also the number of noncanonical states added by state expansion is presented. These results were obtained manually. We have no way to manually obtain the number of useless actions in the action tables for real grammars like PASCAL and Modula-2, since their sizes are too big.

[2]See Appendix A

Grammar	KCT1	KCT2	KCT3	KCT4	PASCAL	MODULA-2
SHIFTs deleted	0	2	9	0	35	45
REDUCEs deleted	0	0	1	0	91	0
states deleted	0	1	4	0	1	0

Table 5.2: Results of the simple algorithm.

Table 2 summarizes the results of the simple algorithm DULA1. The number of deleted useless actions and redundant states for grammars is listed.

Grammar	KCT1	KCT2	KCT3	KCT4	PASCAL	MODULA-2
SHIFTs deleted	2	4	9	1	50	85
REDUCEs deleted	4	0	1	1	266	793
states deleted	2	2	4	1	3	30

Table 5.3: Results of the second algorithm.

Table 3 shows of the results of the second algorithm DULA2 applied for the test suite of grammars.

5.5 Analysis of the Poor Performance for Real Grammars

In this section we will analyze the reason why the two parser-shrinking algorithms worked poorly for character-level grammars of real programming languages. This analysis leads to a modified parser generation algorithm with much better performance. Figure 5.4 gives part of the parser for a character-level grammar called mini_pascal, presented in Appendix A.

In state s_2, the parser expects to see symbols "a", "b", etc. If symbol "b" is shifted from state s_2, it could be reduced to symbol *letter*, or more symbols could be shifted to form the keyword *begin*. To solve the shift-reduce conflict, symbol "e" is noncanonically shifted in

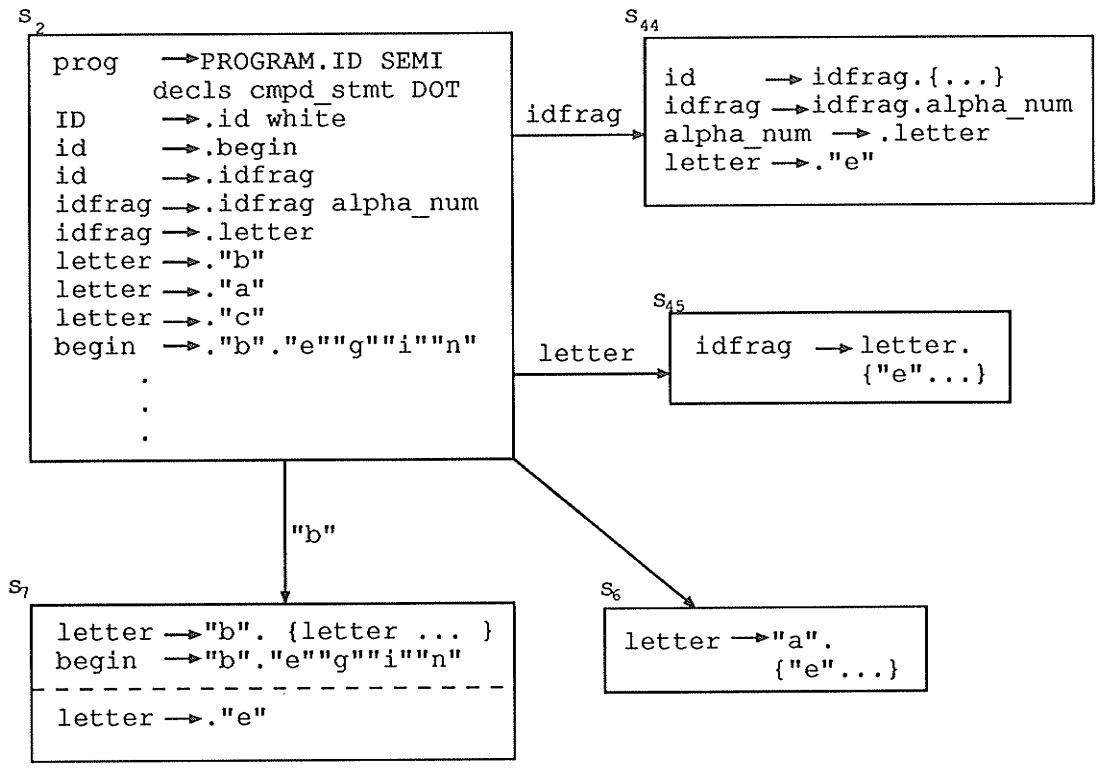

Figure 5.4: Part of a NSLR(1) parser for mini_pascal.

state s_7. In this case, "e" would not be on the lookahead stack when s_{45} is entered, since it would have been reduced to *letter*. Thus it could not serve as a shift lookahead symbol for state s_{44}. On the other hand, if symbol "a" is shifted from state s_2, it will be reduced to symbol *letter* if the next symbol on the lookahead stack is "e". Then the symbol "e" would be needed for both states s_{44} and s_{45}. Thus the shift action $letter \rightarrow \bullet$"$e$" in state s_{44}, and the reduce action $[idfrag \rightarrow letter\bullet, \text{"}e\text{"}]$ have to be kept.

The problem in that for real character-level grammars lookahead symbols that do need to be noncanonically shifted and reduced for some states can be kept unreduced for many other states. The states that do not reduce lookahead terminals, block the elimination of sometimes useless lookahead symbols, because they mean that those symbols will often be

valid.

5.6 Parser Generation with Forced Lookahead Shifting

The above analysis suggests a simple modification to the parser-generation algorithm to increase the number of deleted actions and states. The principle is that if any symbol is deleted from the lookahead set of any reduce action by generating a noncanonical shift action, it should be deleted from the lookahead sets of all reduce actions. We will call this new algorithm NPG_{FS1}, an algorithm for noncanonical parser generation with forced shifts, version 1.

Algorithm NPG_{FS1}: If a lookahead symbol is found to be in conflict in any parser state, consider it to be in conflict in all parser states.

Algorithm NPG_{FS1} does not always work. It generates many new shift actions that cause unresolvable conflicts. The purpose of the added shift actions is simply to help reduce the size of the parser so we can easily eliminate those added actions if they cause new conflicts.

Algorithm NPG_{FS2}: use algorithm NPG_{FS1}, but if forcing conflicts on a particular lookahead symbol causes unsolvable conflicts in the parser then do not force state expansion on that symbol.

Algorithm NPG_{FS2} was implemented by giving feedback to the grammar writer. The grammar writer was informed of which lookahead symbols had caused state expansion in any state, then the grammar writer could code into the grammar which symbols be wanted noncanonically shifted in all states. The grammar writer has the option of not forcing the

shifting of symbols that would cause rejection of the grammar.

For Pascal, Algorithm NPG_{FS2} worked very well eliminating 74 states, which is 7.4% of all states, and almost deleting as many states as were added by noncanonical expansion. For Modula-2 unfortunately no further improvement in the parser tables resulted. It is possible that careful modification of the Modula-2 grammar could lead to better results but such changes were not seriously attempted.

In Tai's original paper, six NSLR(1) grammars were analyzed by hand. For all six grammars the conflict-free noncanonical parser could be made smaller than the basic canonical parser with the conflicts unresolved. These surprising results were not duplicated here. Nevertheless, the excellent performance of eliminating 7.4% of all parser states from a parser for a real character-level grammar offers significant hope for continued research on shrinking noncanonical parsers.

Chapter 6

Conclusion

In this thesis, the structure of $NSLR(1)$ parsers is presented by defining and analyzing several functions. Based on this analysis, algorithms to detect and delete useless actions are presented. These algorithms have been implemented. The shrunken parsing tables for real character-level grammars were verified on a test suite of programs.

The work that would be needed to obtain optimal theoretical and practical methods for NSLR(1) parsers is probably beyond the scope of a master's thesis. This thesis also leads to the invention of a modified parser construction algorithm which generates smaller parsers after eliminating inaccessible states.

Appendix A

Grammar KCT2

!
! Grammar G(2) from the paper "Noncanonical SLR(1) Grammars"
! by Kuo-Chung Tai.
!
S' → <BOF> S <EOF>
S → c A C e | d A D e | A $\bar{A}$ | B $\bar{B}$
A → a
B → a
$\bar{A}$ → f g
$\bar{B}$ → f h
C → d
D → d

Grammar KCT3

!
! Grammar G(3) from the paper "Noncanonical SLR(1) Grammars"
! by Kuo-Chung Tai.
!
S' → <BOF> S <EOF>
S → D E | $\bar{D}$ F
D → d
$\bar{D}$ → d
E → a E b | a b
F → a F | a E

Grammar KCT4

```
!
! Grammar G(4) from the paper "Noncanonical SLR(1) Grammars"
! by Kuo-Chung Tai.
!
S' → <BOF> S <EOF>
S → E | F B
E → b b E | b b
F → b b F | b b
B → b
```

Grammar Mini-Pascal

```
!
! A character-level grammar for a
! Pascal-like mini language.
!
S' →          <BOF> prog <EOF>
prog →        PROGRAM ID SEMI decls cmpd_stmt DOT
id_list →     ID
              | id_list COMMA ID
decls →       decls VAR id_list COLON type SEMI
              |
type →        INTEGER
cmpd_stmt →   BEGIN opt_stmts END
opt_stmts →   stmt_list
              |
stmt_list →   stmt
              | stmt_list SEMI stmt
stmt →        variable ASSIGNOP expr
              | cmpd_stmt
              | ID
              | WHILE expr DO stmt
              | if_then
              | if_then_else
if_then →     IF expr THEN stmt
if_then_else → IF expr THEN stmt ELSE stmt
```

```
variable →      ID
expr_list →     expr
                | expr_list COMMA expr
expr →          term
                | sign term
                | expr ADDOP term
term →          factor
                | term MULOP factor
                factor → ID
                | NUM
                | LPAR expr RPAR
                | NOT factor
MINUS →         "–" white
PLUS →          "+" white
MULT →          "*" white
DIV →           "/" white
COLON →         ":" white
SEMI →          ";" white
COMMA →         "," white
LPAR →          "(" white
RPAR →          ")" white
DOT →           "." white
ASSIGNOP →      ":=" white
sign →          PLUS | MINUS
ADDOP →         sign
MULOP →         MULT | DIV
white →
                | white white_char
white_char →    " "
                | "\ t"
                | "\ n"
                | comment
comment →       cmt_body "}"
cmt_body →      "{"
                | cmt_body any_cmt_char
any_cmt_char →"a" | "b" | "c" | "d" | "e" | "f"
                | "g" | "h" | "i" | "j" | "k" | "l" | "m"
```

| “n” | “o” | “p” | “q” | “r” | “s” | “t”
| “u” | “v” | “w” | “x” | “y” | “z”
| “0” | “1” | “2” | “3” | “4” | “5”
| “6” | “7” | “8” | “9” | “!” | “@”
| “#” | “$” | “%” | “^” | “&” | “*”
| “(” | “)” | “_” | “-” | “+” | “=”
| “ ” | “‘” | “{” | “[” | “]” | “:”
| “;” | “¨” | “”” | “¡” | “,” | “>”
| “.” | “?” | “/” | “|” | “\ ” | “\ t”
| “ ” | “\ n” | “\ f”

BEGIN → begin white

begin → “begin”

DO → do white

do → “do”

ELSE → else white

else → “else”

END → end white

end → “end”

IF → if white

if → “if”

INTEGER → integer white

integer → “integer”

NOT → not white

not → “not”

PROGRAM → white program white

program → “program”

THEN → then white

then → “then”

VAR → var white

var → “var”

WHILE → while white

while → “while”

ID → id white

id → idfrag

idfrag → letter | idfrag alpha_num

letter → “a” | “b” | “c” | “d” | “e” | “f”
| “g” | “h” | “i” | “j” | “k” | “l”

| “m” | “n” | “o” | “p” | “q” | “r”
| “s” | “t” | “u” | “v” | “w” | “x”
| “y” | “z”

digit → “0” | “1” | “2” | “3” | “4”
| “5” | “6” | “7” | “8” | “9”

alpha_num → letter | digit

NUM → digit_string white

digit_string → digit | digit_string digit

Bibliography

[Tai79] Kuo-Chung Tai, "Noncanonical SLR(1) Grammars." *ACM TOPLAS*, Vol. 1, No. 2, page 295-320 (Oct. 1979).

[DeRe82] Frank DeRemer and Thomas Pennello, "Efficient Computation of LALR(1) Look-Ahead Sets". ACM TOPLAS, Vol.4, No.4(1982), 615-649.

[Aho86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools". Addison Wesley, Reading, Mass. (1986)

[Bake81] Theodore P. Baker, "Extending Lookahead for LR Parsers". J. Comp. Sys. Sci. 22(1981), 243-259

[Knuth65] Donald Knuth, "On the Translation of Languages From Left to Right". Inf. Control 8 (1965), 607-639.

[DeRe71] Frank DeRemer, "Simple LR(k) Grammars". CACM 14, 7 (1971), 453-460.

[Culik] K. Culik and R. Cohen, "LR-regular grammars-an extension of LR(k) grammars". J. Comp.Sys.Sci. 7(1973), 66-96.

[Floy64] R.W.Floyd, "Bounded context syntactic analysis". CACM 7,2 (1964), 62-67

[Park85] Joseph C.H. Park, K,M. Choe and C.H. Chang, "A New Analysis of LALR Formalisms". ACM TOPLAS, Vol.7, No.1 (Jan. 1985), 159-175.

[Sal89A] Daniel J. Salomon, "Metalanguage Enhancements and Parser-Generation Techniques for Scannerless Parsing of Programming Languages". Ph.D. Thesis, University of Waterloo. Tech. Report CS-89-65 (1989).

[Sal89B] Daniel J. Salomon and Gordon V.Cormack, "Scannerless NSLR(1) Parsing of Programming Languages". ACM SIGPLAN Notices, (June 1989), 170-178.

[Syzm76] T.G. Syzmanski and J.H. Williams, "Noncanonical Extensions of Bottom Up Parsing Techniques", SIAM J. Computing 5, 2 (June, 1876), 321-350.

[Col70] A. Colmerauer, "Total Precedence Relations". J.ACM 17, 1(Jan. 1970), 14-30.

[Will75] J.H. Williams, "Bounded Context Parsable Grammars", Information Control 28, 4(1975), 314-334.