Metalanguage Enhancements and Parser-Generation Techniques for Scannerless Parsing of Programming Languages

Daniel Joseph Salomon

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Research Report CS-89-65
December, 1989

Metalanguage Enhancements and Parser-Generation Techniques for Scannerless Parsing of Programming Languages

by

Daniel Joseph Salomon

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1989

ABSTRACT

In current compiler-construction techniques, it is standard practice to split syntax analysis into two phases: scanning and parsing. Some disadvantages of two-phase syntax analysis are presented. Two-phase syntax analysis is in standard use principally due to the inadequacy of the commonly-used metalanguages for completely, unambiguously, and concisely describing a programming language in a single grammar, and the inadequacy of the commonly-used parser-generation techniques for processing complete descriptions.

This thesis proposes a notation, called *restricted context-free grammars (RCFG's)* that allows the syntax of a programming language to be completely described by a single, succinct, and unambiguous grammar. Such grammars are complete in the sense that even the syntax of comments and of white space are described, and the terminal symbols are all single input characters. Parser generation techniques are presented for this new notation, and since the grammars are complete, no scanner phase is needed.

Restricted context-free grammars consist of productions, as do ordinary context-free grammars, and of two new restrictive rules: the exclusion rule, and the adjacency-restriction rule. The intended use of RCFG's is to prepare a concise, but ambiguous grammar using productions, and to resolve the ambiguities using the restrictive rules. The new rules are formally defined and their properties analyzed.

Concise single-phase grammars require a powerful parsing scheme. The noncanonical SLR(1) parser generation method of Tai can meet these needs. An overview of NSLR(1) parsers and parser generation is given. A correction is presented for a known error in the published version of the parser generator, and enhancements are described to allow the processing of the new notation.

The efficiency of single-phase parsers is evaluated, and it is shown that such parsers can have comparable efficiency to the traditional two-phase parsers.

An NSLR(1) parser generator that accepts the new rules has been implemented and tested on a complete single-phase grammar for ISO Pascal and for Modula-2. The grammar for Pascal is presented as an appendix.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisor, Gordon Cormack for his guidance and constructive criticism in the preparation of this thesis. I would also like to thank the other members of my thesis committee, Don Cowan, Charles Colbourn, Lee Dickey and Nigel Horspool for their help.

This work was principally funded by grants from the *Natural Sciences and Engineering Research Council of Canada*, and the *Information Technology Research Centre of Ontario*.

DEDICATION

To the memory of my father who introduced me to science early in my youth, and to my mother who cleaned up after the series of acid burns, short circuits, minor fires, and explosions that followed.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Conventional wisdom tells us that when writing a compiler we should split parsing into two phases: lexical analysis by a finite-state scanner and syntax analysis by a pushdown parser. Unfortunately both compiler writers and compiler-compiler writers suffer in this scheme. The compiler writer suffers in that he must partition the grammar for the programming language that he is implementing into two interelated grammars, and he must design an interface between the two phases. In addition, it is not always clear how much semantic analysis should be done in the scanner; specifically how much processing of literal constants should be done. (In attribute grammars this problem corresponds to devising a way to pass attributes between the two grammars.)

To clarify the deficiencies of this approach, consider a programming environment where no single programming language has adequate power to solve any problem. Every problem must be solved using two different programming languages, and every programmer must keep at least two language reference manuals at hand. Few programmers would relish such an environment, and yet compiler writers are expected to accept it.

The designer of a compiler-compiler that uses a two-phase approach also faces difficult decisions. He must choose what metalanguages are to be used by the scanner generator and the parser generator, implement two separate automata generators, design the form of the interface between them, and prepare user documentation for the two metalanguages and the interface.

Nevertheless, two-phase syntax analysis is in standard use principally because of two distinct problems. The first problem is the inadequacy of the commonly-used metalanguages for completely, unambiguously, and concisely describing a programming language in a single grammar. The second problem is the inadequacy of the commonly-used parser-generation techniques for processing complete grammars. It is these two problems that this thesis addresses.

1.1. The Pervasiveness of Character-Level Grammars

The style of modern language-description manuals attests to the desirability of describing a programming language with a single grammar. It is now common for programming-language definition manuals to include a *character-level grammar* that describes the language being defined right down to single input characters. A character-level grammar is characterized by EBNF rules such as:

Letter	→ "a" | "b" | "c" | ... | "z"
Digit	→ "0" | "1" | "2" | ... | "9"
Identifier	→ *Letter* { *Letter* | *Digit* }
DigitSequence	→ *Digit* { *Digit* }
UnsignedInteger	→ *DigitSequence*
UnsignedReal	→ *UnsignedInteger* "." *DigitSequence* ["e" *ScaleFactor*]
	| *UnsignedInteger* "e" *ScaleFactor*
ScaleFactor	→ [*Sign*] *UnsignedInteger*
Sign	→ "+" | "−"
StringElement	→ "' '" | any_character_except_apostrophe
CharacterString	→ "'" *StringElement* { *StringElement* } "'"

These sample rules were taken from a grammar for Pascal,[26] but similar rules can be found in grammars for PL/I,[6] Modula-2,[59] and Ada.[16]

1.2. Common Deficiencies of Character-Level Grammars.

Despite their apparent attention to detail, character-level grammars have a number of typical deficiencies:

1) They are incomplete. There are a number of syntactic features typically missing from the character-level grammars:
   - The syntax of comments is not given.
   - The permitted use of white space (blanks, tabs, newlines, and comments) is not given. For instance, the common Pascal grammars do not specify whether blanks may appear in a record field selector. Can the selector "`employee.name`" appear as "`employee . name`," or split across lines? (In usual implementations it can.) Similarly they do not specify if blanks may appear in floating constants. Can the constant "`1.86e5`" appear as "`1.86 e5`"? (In usual implementations it cannot.)
   - The required use of white space is not given. For instance, white space must appear between keywords and identifiers or the keyword is to be treated as part of the identifier.

   This missing information is usually supplied in English by supplementary documentation, even though a CFG is powerful enough to supply it.
2) They are ambiguous. An examination of the ISO grammar for Pascal shows the common kinds of ambiguity.
   - The description of the nonterminal *Identifier,* as shown above, includes all Pascal keywords, even though they are reserved and are supposed to be excluded.
   - When reserved words are embeded in identifiers, there may be more than one valid parse of a sentence. For instance, according to the ISO grammar for Pascal, the program fragment

     ```
     BEGINWORKEND;
     ```

     can be parsed as either as a compound statement invoking the procedure `WORK`, or as a simple statement invoking the procedure `BEGINWORKEND`. We call this kind of ambiguity a *longest-match ambiguity.*
   - The grammar contains the so-called *dangling-else* ambiguity. The program fragment

     if $B1$ **then**
     if $B2$ **then**
     $S1$
     else
     $S2$;

     can be parsed so that $S2$ will be executed when $B1$ is true and $B2$ is false, or when $B1$ is false.
3) They are unsuitable for the common automatic-parser-generation methods. They require parsers with greater power than is available from an LL(1) or LR(1) parser. Unfortunately, the common parser generators typically provide even less power than an LR(1) parser. The UNIX tool YACC, for instance, generates LALR(1) parsers.

1.3. Reasons for Deficiencies of Character-Level Grammars

Why are grammars published with so many deficiencies? The three desirable properties of published grammars are that they be:

1) **correct**, which implies that they be precise, complete and unambiguous,
2) **human readable**, so that implementors and programmers can understand them, and
3) **suitable for automatic parser generation**, so that reliable compilers can be easily written.

These goals often conflict. A complete and unambiguous context-free grammar would be too long to be human readable. Similarly, the changes to a grammar needed to make a grammar LALR(1) tend to increase the length of the grammar. Excessively long grammars also tend to impede automatic parser generation, as most automatic parser generators produce at least one state per rule, and thus long grammars generate large parsers. The conclusion one would draw from this analysis is that published grammars have sacrificed correctness and machine processability for conciseness in order to increase readability.

A compiler writer attempting to prepare a grammar that meets the three objectives given above suffers from the constraints of context-free grammars. Context-free grammars have only one type of rule, the production, and only one operator, the alternation bar. This simplicity has aided in the description of the formal properties of context-free grammars, but impedes the concise description of programming languages. Some attempts have been made to enhance the notation of context-free grammars to facilitate programming-language description. The best known of these are the enhancement of BNF notation into EBNF. EBNF has the added features of square brackets for optional phrases and curly braces for repeated phrases. Tests by the author on grammars for Pascal, Modula-2, and Ada have shown that due to these simple enhancements the same grammar can be expressed in EBNF using from 20% to 43% fewer rules than BNF. These improvements, however, are not enough for complete character-level grammars.

To avoid these difficulties, compiler writers often abandon the use of a grammar for describing the scanner phase and code it directly in an implementation language. By doing this they lose the benefits of using a grammar. An implementation usually expresses the syntax of a language less clearly than does a grammar. This was one of the reasons for inventing grammars.

This thesis presents a metalanguage that allows a correct description of programming languages, while retaining readability. This thesis also presents a parser construction algorithm that can handle the proposed metalanguage.

1.4. Proposed Additions to CFG Notation

Currently, context-free grammars have only one rule, the production, and this rule has no negative form. In other words, one can describe a symbol in terms of what it generates, but there is no mechanism for describing what it does not generate, even though a negative description may be shorter and clearer. Our proposal for a more powerful metalanguage consists of enhancing context-free grammars with two new restrictive rules, the exclusion rule, and the adjacency restriction.

The technique proposed is to write compact but ambiguous context-free grammars, augmented by restrictive rules that disambiguate them. Grammars that use these restrictive rules are called *restricted context-free grammars* (RCFG's), and in particular, a BNF grammar that uses these new rules is called a *restricted BNF* (RBNF) grammar. To rewrite such grammars using only productions would require awkward grammar transformations, often resulting in an exponential size increase (thousands more rules for Pascal-like languages), and convolution of the grammar's structure so as to compromise its utility for semantic analysis and translation.

The two new rules are precisely defined, in Chapter 5, by:

1) Giving a formal description of how they affect derivations.
2) Giving grammar transformations that produce a pure CFG from a restricted CFG.
3) Giving algorithms for generating parsers from restricted CFG's. Since the grammar transformations of item 2 tend to produce very large grammars, these parser generation algorithms work without applying the transformations.

The new rules have additional practical value for parser generation. Some unambiguous grammars that do not use the restrictive rules may require k symbol lookahead, where $k > 1$, by an LR parser. With the addition of the restrictive rules, even though the grammar is already unambiguous, it may be possible to parse the language generated by the grammar using a single symbol of lookahead by an LR parser.

1.5. Parser Generation for Restricted Context-Free Grammars

To supply the parsing power needed for character-level grammars a parser generator based on the noncanonical SLR(1) method of Tai;[55] is proposed. His method generates two-stack deterministic parsers. The greater power of his method is derived from the fact that multiple lookahead symbols can be reduced to a higher-order nonterminal, which is then used as the lookahead symbol.

Tai's parser-generation method cannot be used exactly as published. It requires three principal improvements. First, an error in the computation of lookahead sets must be corrected. Second, the handling of ϵ-productions must be improved. This change is necessary since the grammar symbol that generates white space is ubiquitous in a character-level grammar, and it can generate ϵ. Third, the handling of the new restrictive rules must be incorporated into the method.

1.6. Conclusions and Results

There are two principal conclusions of this thesis. The first is that restricted context-free grammars can be used to prepare complete, concise and unambiguous descriptions of modern programming languages, and the second is that such grammars can be processed by automatic parser generators. To demonstrate the first conclusion RCFG grammar for Pascal is presented in an appendix. The second conclusion is demonstrated by the algorithms given in this thesis for processing a class of RCFG grammars called SE-SAR-NSLR(1) grammars (the abbreviation is explained in the next paragraph), and the actual implementation of the algorithms. Note that the first conclusion is independent of the second. That is, RCFG's are useful for describing programming languages, even if traditional methods are used to implement the parser.

The abbreviation SE-SAR-NSLR(1) stands for *simple-exclusion, simple-adjacency-restriction, noncanonical simple LR(1)*. Generating parsers for grammars that make unrestricted use of exclusion rules and adjacency-restriction rules is a very difficult problem. We have therefore proposed a set of limitations on the use of these rules that permit the generation of parsers using the kind of simple treatment of lookahead sets that gives *simple* LR parsers their name. SLR(1) parsers are called simple because the consistency of the parser can be tested by examining each state of the parser individually without considering the paths between states. Our limited class of grammars still provides enough power to prepare complete, unambiguous grammars of a reasonable length for modern programming languages.

Appendix D presents an SE-SAR-NSLR(1) grammar for Pascal. This grammar is complete and unambiguous, and with 563 rules is still concise enough to be readable by humans and to generate a parser of reasonable size. Appendix F is a user's manual for our implementation of an SE-SAR-NSLR(1) parser generator.* This parser generator was tested on numerous RCFG's including grammars for ISO Pascal and Modula-2, and the generated parsers were verified on a test suite of Pascal and Modula-2 programs prepared for CS-444, a compiler-construction course given at the University of Waterloo. The Modula-2 parser was also tested on the source code of the parser generator itself. The parser generator also provides the basis for a Modular Attribute Grammar system.[18]

The contents of this thesis have been summarized and presented at the SIGPLAN '89 Conference on Programming Language Design and Implementation.[49, 50]

* This parser generator was based on an SLR(1) parser generator written by Gordon Cormack.

1.7. Preview of Remaining Chapters

Chapter 2 gives an overview of existing work by other researchers that is related to this thesis. That chapter has two principal topics of interest: (1) enhanced parsers and parser generators that can provide the power needed to parse character-level grammars, and (2) methods of resolving ambiguities in context-free grammars.

Chapter 3 describes attempts to use existing compiler-construction tools to process single-metalanguage descriptions of the programming language Pascal. The objective of this chapter is to show the difficulties that a compiler writer faces when taking this approach. This endeavor also showed that single-metalanguage translators can have processing efficiencies similar to dual-metalanguage translators.

Chapter 4 describes the noncanonical SLR(1) parsing strategy of Tai. An error in Tai's original parser construction algorithm is described and a correction proposed. Enhancements are also proposed that permit the processing of character-level grammars.

Chapter 5 presents detailed descriptions of the two new proposed rules for context-free grammars, exclusion rules and adjacency-restriction rules. Formal definitions for these new rules are given, and strategies for incorporating them into existing LR parser generators and Tai's Noncanonical SLR parser generator are presented.

Chapter 6 presents a summary of the results of this thesis, and proposes future research toward accepting a larger class of restricted CFG's and generating smaller parsers.

Chapter 2

Related Work

In this chapter, existing work related to parsing complete character-level grammars is surveyed. There are two principal topics of interest: parsing techniques powerful enough for processing complete character-level grammars for programming languages, and methods of disambiguating programming-language grammars.

2.1. Problems Parsing Character-Level Grammars

Before presenting advanced techniques suitable for parsing character-level grammars, we give an example, taken from Pascal, of the kind of parsing problem that necessitates advanced parsing techniques. Consider the following grammar rules:

VarDeclPart → *VAR VarDeclList*
IdList → *Identifier* | *IdList* "," *Identifier*
VarDecl → *IdList* ":" *Type*
VarDeclList → *VarDecl* ";" | *VarDeclList VarDecl* ";"

and the following two fragments of Pascal code:

```
    ...
var
    A : real;
    I : integer;
procedure P (x : real);
    ...
```

```
    ...
var
    A : real;
    I : integer;
    proceeds : real;
    ...
```

A parser processing these two program fragments from left to right would perform identical actions until the "p" of "procedure" or the "p" of "proceeds" is encountered. Up until that point the strings "A : real;" and "I : integer" have been recognized as variable declarations. On seeing the "p" of "procedure" in the first program fragment, the parser should end the recognition of *VarDeclList*, and begin recognition of a procedure declaration. In the second fragment, on the other hand, the parser should recognize that the variable declaration list has not ended, and should simply begin the recognition of another variable declaration. Whether the parser is controlling the immediate production of code, as in one-pass syntax-directed code generation, or is controlling the construction of a parse tree, these two choices have significantly different results.

If a parsing technique is being used that uses a single symbol of lookahead information, as is the case with LL(1) and LR(1) parsers and their derivatives, then the only lookahead information available would be the single character "p". This is simply not enough information to make the parsing decision for a parser generated from the given grammar fragment. A parser would actually have to lookahead to the second "e" in "proceeds" before it could tell the two code sequences apart. In general a parser would have to look all the way to the blank following "procedure" before it could be sure that a *ProcedureDeclaration* was indeed starting, which means that ten symbols of lookahead information would be needed.

The obvious solution of using an LR(k) parser (or LL(k) parser) with a large enough value of k is simply not practical. The cost of increasing k grows disproportionately with the benefit. When a parser for a character-level grammar is being constructed there are up to 128 terminal symbols, and hence there could be up to 128^k lookahead strings for each state of the parser. For $k = 10$ this would

require about 10^{21} bytes of storage per state, a figure that exceeds even the size of the largest existing secondary storage devices. One may justifiably argue that it should be possible to find patterns in these lookahead strings, and use considerably less storage by storing only some compact representation of those patterns, and devise a machine for interpreting those patterns. In fact, the XLR and LAR parsers described in Sections 2.3.4 an 2.3.5 below can be viewed as using this technique. Nevertheless, the straightforward construction of an LR(k) parser, for k as large as 10, is not practical.

In fact, it has been shown that any LR(k) grammar can be rewritten as an LR(1) grammar. In addition, grammar rewriting is often employed and required for making published grammars acceptable to existing parser generators. But the grammar rewriting for the above grammar segment to generate an LR(1) parser whose lookahead symbols represent a single character of input, would totally destroy the readability of the grammar. Furthermore, the author speculates that the great width of the LR(k) tables would simply be exchanged for LR(1) tables of great length. That is, a large number of lookahead possibilities would be exchanged for a large number of parser states, leading to similar parse table sizes.

2.2. Traditional Multiphase Syntax Analysis

In order to implement parsers for character-level grammars, the traditional practice has been to add one or more preprocessing phases before the parsing phase. This technique is illustrated in Figure 2.1.

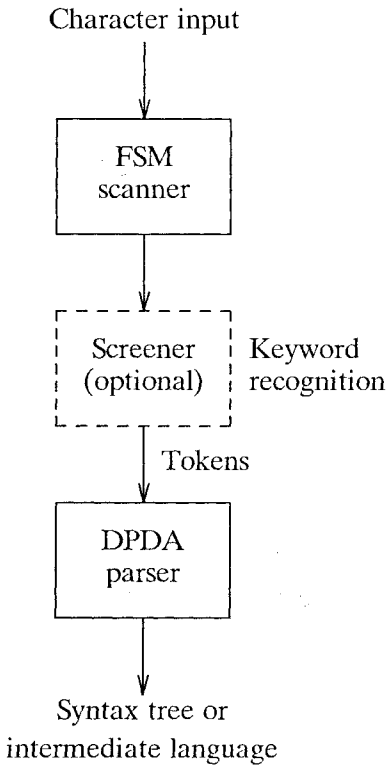

Figure 2.1: The traditional method of implementing syntax analysis.

In the syntax-analysis* method shown in Figure 2.1, a finite-state machine preprocesses input for a DPDA parser. In addition, some implementations use an intermediate phase called a *screener* to

* In the literature, it is common to call the processing done by the scanner phase *lexical analysis*, and to treat it as distinct from syntax analysis. That distinction is not made here, and our use of the term *syntax analysis* includes lexical analysis.

recognize reserved keywords, rather than having them recognized by the scanner. The parser receives tokens, rather than single characters, as input, where each token represents strings of one or more input characters.

2.2.1. Advantages of Multiphase Parsing

There are reasonable arguments for using multiphase syntax analysis. By using the multiphase method, instead of implementing a complex, multisymbol-lookahead parsing strategy and its parser generator, the parsing problem is solved by partitioning it into two or more simpler modules. Modularization of a program is generally perceived as being good.

Another perceived advantage of the multiphase method is that a finite-state preprocessor for a DPDA parser should run more efficiently that a DPDA parser processing raw input.

2.2.2. Disadvantages of Multiphase Parsers

Unfortunately there are serious disadvantages to multiphase syntax analysis. The first is that the grammar for the language to be implemented must be partitioned into at least two parts: a high-level grammar for the parser and a low-level grammar for the scanner. Most modern languages are designed so that their grammars can be partitioned in this way, but this need not be the case.

Second, if a finite-state machine is used as the scanner, the parts of the partitioned grammar have to be written in different metalanguages. This is so because a finite-state preprocessor cannot, in the general case, recognize a language described by a context-free grammar, and regular expressions are not powerful enough to describe the syntax of the common programming language. This fact is the reason that the UNIX scanner generator LEX, and parser generator YACC use different metalanguages for their input grammars.

The third disadvantage to multiphase-syntax analysis is that it requires the design of an interface between the phases. The interface can be fairly complex, since it must describe both the token being transmitted, and the attributes of the token. Token attributes are things like the name of an identifier token, and the values of string or numeric constants. A language may be required for describing the interface. For instance, the interface between LEX and YACC is described by files in the C programming language.

The apparent advantages of multiphase parsers can also be questioned. The first apparent advantage was modularization. Even though a multiphase parser has been divided into modules, these modules are usually *closely coupled*.[44] They are closely coupled in the sense that the two modules can seldom be changed independently. Changes to one module often require corresponding changes to the other module. When modules are closely coupled, modularization can actually increase the difficulty of maintaining the system rather than decreasing it. In addition, the modularization used can be categorized as phase modularization, which Parnas warns against in his classic paper on modularization.[44]

It should be pointed out that the grammar for a programming language can be modular, even though its parser is not. Indeed most character-level grammars presented in the programming language manuals are highly modular. Not only do these grammars group nonterminals with similar purposes into recognizable regions of the grammar, but also the description of each nonterminal can itself be thought of as a subgrammar. Work by Heering, Klint, and Rekers[23] on incremental generation of parsers may even lead to separate compilation of grammars modularized in this way.

The second apparent advantage was an expected increase in efficiency due to using a finite-state preprocessor for the push-down automaton. The increase in efficiency is not necessarily realized, either in theory or in practice. Theory tells us that an FSM and a DPDA both run in linear time on the length of the input, so that the performances can only differ by a constant factor. Further analysis presented in Chapter 6 shows that this constant is small, and may actually favour a single-phase DPDA in some cases. Actual tests, presented in Chapter 3, show that a DPDA scanner, prepared using YACC, has a similar efficiency to an FSM scanner, prepared by LEX.

It should also be pointed out that multiphase parsers and multi-metalanguage grammars do not resolve any of the ambiguities mentioned earlier; they solve only lookahead problems. Every ambiguity that existed before the grammar was partitioned will still exist after the grammar is partitioned.

2.3. Existing Work on Single-Phase Parsing or Unified Metalanguages

Other researchers have proposed methods for single-phase parsing, or for constructing parsers that may be suitable for parser generation using an unpartitioned grammar, or at least using a single metalanguage. In this section we shall consider some of these. In order to keep parsing efficiency comparable to the traditional multiphase method, only parsing methods that run in at least linear time on the length of the input are considered.

2.3.1. Scannerless Parsing

One of the earliest references to scannerless parsing is given by Mavaddat[39] He shows how operator-precedence parsing can be extended to avoid the need for prescanning numeric input into tokens. His application is in desk calculator programs, where all identifiers are single letters. His technique could be extended to avoid the scanning of identifiers too, but operator-precedence parsing is generally considered not to be powerful enough to parse modern programming languages.

2.3.2. Partitioned Grammars Using a Single Metalanguage

DeRemer[14] proposed that the scanner and the parser could be described by the same metalanguage, a context-free grammar, while keeping a multiphase parse. His method uses an LR parser generator for both phases. Since his proposal still involved multiphase syntax analysis, the problem of interfacing the phases still exists, and he does not address this issue. His handling of ambiguities is discussed in Section 2.4.3 below.

DeRemer's proposal does not decrease the complexity of the parser, but it simplifies the parser generator. The same parser generator module can be used to generate both the scanner and the parser. It also simplifies the preparation of the partitioned grammars, since the same metalanguage is used.

2.3.3. Automatic Grammar Partitioning

Krzemien and Lukasiewicz[33] discuss the automatic extraction of the FSM scanner phase from a unified grammar. They present an algorithm for extracting regular subgrammars from a context-free grammar. By their method, the compiler writer prepares a single grammar for the language being implemented, and the parser generator automatically partitions it into a regular grammar for the scanner phase, and a context-free grammar for the parsing phase. They do not, however, discuss how to deal with the common ambiguities in programming-language grammars.

This method simplifies the compiler writer's task in that he now deals with a single grammar, and a single metalanguage. The method however leads to quite a complex parser generator comprised of at least three modules: a grammar splitter, a scanner generator and a parser generator.

2.3.4. Regular-Right-Part Grammars and Parsers

LaLonde[35,36,37] proposes the use of regular right part (RRP) grammars, parsers and parser generators. In his method, the right part of a production rule may be a regular expression. This approach results in highly expressive grammars. Sample rules from his metalanguage would look like the following:

$$A \rightarrow a(C \mid c)$$
$$B \rightarrow b \mid a^*Bc$$

He presents a parser generation algorithm that can accept RRP grammars and produce RRP parsers.

An RRP parser has six actions: **readahead**, **readback**, **shift back** *n* **goto**, **shift back** *n* **reduce**, **accept**, and **error**. The extra actions allow the parser to change state without modifying the parse stack. In this way a finite-state machine can perform regular-expression recognition interleaved with a DPDA parse. Thus the finite-state scanner is not eliminated, but rather is incorporated directly into the parser.

LaLonde explicitly states that his objective is to unify the description of scanners and parsers. He shows that his parsing method sometimes runs more efficiently than a parser generated from a pure CFG description of the same language. He does not, however, discuss the treatment of the common ambiguities appearing in programming language grammars.

2.3.5. Extended Lookahead LR (XLR) Parsers

Baker[7] presents a practical parsing technique that can make use of more than a single symbol of lookahead. First an ordinary LALR(1) parser is constructed. If any conflicts remain that are not resolved with a lookahead of a single terminal symbol, then a nondeterministic finite-state machine processes additional terminal symbols of lookahead until all but one of the conflicting parser actions is eliminated. He postulates that a deterministic version of his conflict-resolution scheme would result in excessively large parse tables.

Baker does not mention the parsing of character-level grammars as an objective of his parsing method. Instead he proposes to solve difficult parsing problems in the grammars of Pascal and Ada that are normally solved by rewriting the grammar. He also suggests that his parsing method facilitates error recovery.

We postulate that Baker's method is powerful enough to generate parsers from character-level grammars, once a suitable treatment for the common ambiguities is devised. In such an application, however, conflict resolution would be invoked regularly, and the nondeterministic nature of the conflict-resolution strategy would significantly affect the efficiency of the parser, so that it would not necessarily run in linear time. In the applications cited by Baker, on the other hand, conflicts are significant but rare, and the efficiency of conflict resolution hardly affects total parsing efficiency.

2.3.6. LAR Parsers

Bermudez & Schimpf[9] present a parse-table generator for a parser that they call LAR(*m*).* They build an LR(0) parser, and, like Baker, resolve any conflicts by running a finite-state automaton on the lookahead string, but unlike Baker their FSA is deterministic.

An LAR(*m*) parser has seven parser actions:

(1) **shift** *q*,
(2) **reduce** *n*,
(3) **accept**,
(4) **error**,
(5) **lookahead-scan** *q'*,
(6) **lookahead-shift** *q*, and
(7) **lookahead-reduce** *n*.

The last 2 actions are performed by a lookahead automaton when resolving a conflict. A successful conflict resolution consists of a series of **lookahead-scan** *q'* actions, representing transitions between states of the lookahead automaton, followed by a lookahead-shift *q* or a **lookahead-reduce** *n*. A **lookahead-shift** *q* action indicates that the parser should resolve the original conflict by shifting and going to state *q*, and a **lookahead-reduce** *n* action indicates that the parser should resolve the conflict by a reduction on rule *n*.

The states of the lookahead automaton are computed based on the set of states that the inconsistent LR(0) parser could enter. As such the finite-state lookahead automaton is simulating the action of the LR(0) parsing automaton, but with a finite stack size of *m*.

---

* The letters LAR are not an acronym; they are a pure fabrication.

This parsing method is superior to Baker's in that the lookahead automaton is deterministic; nevertheless it does have some drawbacks. After looking ahead at an arbitrary length of input, the lookahead automaton resolves only one parsing conflict, and after the resolution any input scanned must be reprocessed through the actual LR(0) automaton. This rescanning wastes some parsing work done during the lookahead parser simulation.

Another limitation of the method is that although the lookahead automaton is simulating the LR(0) parser, it is still only a finite-state machine. As a result the lookahead set that can resolve a conflict is still regular.

Another problem is that a separate lookahead automaton is built for each state in conflict. If a lookahead symbol for state *A* would lead the parser to another conflicting state *B* then the lookahead automaton for state *B* would be duplicated entirely in that of state *A*. Furthermore, once the conflict in *A* has been resolved, the parser would be directed to state *B* where another conflict exists and another conflict resolution automaton would be invoked. The conflict resolution automaton for state *B* would reprocess some of the same lookahead information that was processed for state *A*. Bermudez and Schimpf do not discuss how this duplication of conflict resolution automata and reprocessing of input could be eliminated or minimized.

These disadvantages did not concern Bermudez and Schimpf in that they were interested mostly in grammars that are almost LR(0) consistent. The motivation for their work was in processing grammars generated by automatic grammar rewriting systems where the user does not always have full control over the grammar being processed. They give further uses for their method by showing how their parser generator can be used to solve a difficult problem in parsing PL/I.

2.3.7. Noncanonical SLR Parsers

Szymanski and Williams[54] present the theory of noncanonical bottom-up parsers that can make nonleftmost reductions on sentential forms. Their modification can be made to almost any existing bottom-up parser to yield a parser that can accept a larger class of languages while keeping linear time characteristics.

Tai[55] presents a parser-generation method for noncanonical SLR(1) parsers, called NSLR(1) parsers for short, that use Szymanski's noncanonical parsing principles. Tai's parser uses two parsing stacks and is schematically illustrated in Figure 2.2.

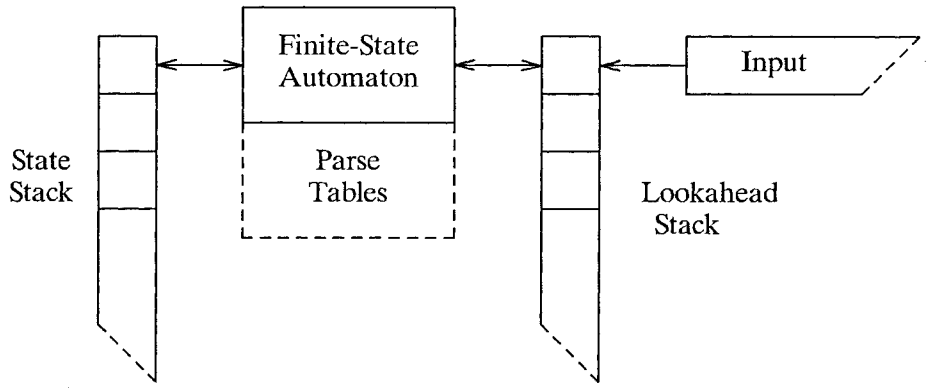

Figure 2.2: Schematic representation of an NSLR(1) parser. The principle differences from an SLR(1) parser are the addition of a lookahead stack, and the redefinition of the **reduce** action to make use of that stack.

The two stacks of the parser are called the *state stack*, which is used like a standard DPDA stack, and the *lookahead stack*, which is the added stack. (Tai calls the lookahead stack the *symbol stack*.) The lookahead stack initially contains the input characters, with the first character at the top of the stack. (In actual implementations, input data can be made to appear as if it is initially loaded on the stack without actually performing any push operations to initialize the stack.) A symbol pushed back onto the lookahead stack, during parser execution, will appear to the parser to be the next input symbol.

Tai's parse-table construction method method begins by constructing SLR(1) parse tables. If the parse tables contain a conflict on some lookahead symbol, the conflict is resolved by shifting the lookahead symbol (and possibly further right context) onto the state stack where, using further lookahead information, it may be reduced to a higher-level nonterminal. That reduced symbol will then be pushed back onto the lookahead stack where it will serve as the new lookahead symbol. Since the new lookahead symbol may have been reduced from many lower-level lookahead symbols, and since further right context was used in its reduction, the new lookahead symbol can often resolve the original conflict. A complete description of NSLR(1) parsing and parse-table generation is given in Chapter 4.

The changes to an SLR(1) parser to obtain an NSLR(1) parser are minimal. An NSLR(1) parser has only the same four parser actions of any LR parser: **shift**, **reduce**, **accept**, and **error**. The only difference in the parser actions is that the **reduce** action is redefined so that it pushes the reduced symbol back onto the lookahead stack. In addition, the parse tables are simplified by eliminating the GOTO table. An NSLR(1) parser can directly execute any LR(1) parse tables by changing every entry in the GOTO table into a **shift** action, and adding it to the action table. This is possible because a GOTO operation actually represents a shift from a temporary location used to store a single reduced symbol. Since the **reduce** action has been changed to make the reduced symbol look like the next lookahead symbol, a **shift** action behaves just as a GOTO operation would have.

Since an NSLR(1) parser processes the lookahead information with a PDA, the lookahead language, the set of lookahead strings that can resolve a conflict, can be a context-free language. In this way NSLR(1) parsers are more powerful than any of the above methods, whose lookahead languages are all regular sets. In fact, Tai shows that an NSLR(1) parser can recognize even some nondeterministic context-free languages.

2.3.8. The Proposed Parser Generation Scheme

Our proposal is to describe completely the syntax of a programming language using a single character-level grammar and to use an advanced lookahead parser to generate a single phase parser from that grammar. We chose Tai's Noncanonical SLR parser generation method as the basis of our parser generator. This choice was made for several reasons:

1) The complexity of an NSLR(1) parser, and the NSLR(1) parser generator was judged to to be the least of the other techniques described above. In some ways the complexity of an NSLR(1) parser generator is even less than that of an SLR(1) parser, for although it has an extra stack, its parse tables are more uniform, having no distinction between GOTO tables and action tables.
2) The set of languages accepted by an NSLR(1) parser is a proper superset of the set of languages accepted by the other techniques described above.
3) Although the above methods can accept some grammars that are not NSLR(1), the NSLR(1) grammars are adequate for describing real programming languages.

To make NSLR(1) parsers useful for parsing single-phase grammars, some corrections and enhancements to Tai's algorithm were needed.

1) The lookahead sets of all states must be computed to include nonterminals as well as terminals. In the published algorithm, only lookahead sets for expanded states contain nonterminals, which produced incorrect parsers for some grammars.

2) The state expansion algorithm must be improved to include only essential ϵ-reducing items. Tai's simpler algorithm generates too many items, introducing spurious conflicts.
3) Special treatment is needed for *invisible* symbols, symbols that cannot produce terminal symbols, either directly or indirectly.

A complete description of the NSLR(1) parser and parser generation algorithms are presented in Chapter 4, along with details of the correction and enhancements made to the algorithms.

2.4. Previous Work on Disambiguation

As was pointed out earlier, the traditional two-phase parsing method does not in itself resolve any grammar ambiguities. All grammar ambiguities that existed before the grammar was partitioned will still be present after partitioning, in either the scanner grammar or the parser grammar. Ambiguities transferred to the parser grammar, such as the dangling-else ambiguity, are usually resolved by rewriting the grammar to resolve the ambiguity, or by doctoring the generated parser. Ambiguities transferred to the scanner grammar usually require individual attention of the compiler writer.

2.4.1. Disambiguation by Grammar Rewriting

The traditional way of resolving ambiguities in a grammar is to rewrite the grammar to eliminate the ambiguities without changing the language described. The problem with this approach is that context-free grammars contain only one type of rule, the production, and this rule has no negative form. This means that a grammar that could be described quite concisely with a negative rule such as the statement, "The symbol *Identifier* **does not** generate any of the reserved keywords," must be rewritten with a larger number of rules explicitly describing what *Identifier* **does** generate.

Consider for example how a Pascal grammar would be rewritten to resolve the reserved-keyword ambiguity. The ambiguous EBNF description of *Identifier* would take the form:

Identifier → *Letter* { *Letter* | *Digit* }

To rewrite this definition unambiguously would require rules such as the following:

Identifier → *UnreservedWord* { *Letter* | *Digit* }

UnreservedWord → *h* | *j* | *k* | *q* | *x* | *y* | *z*
| *a Digit*
| *a a* | *a b* | *a c* | ... | *a m*
| *a n Digit*
| *a n a* | *a n b* | *a n c*
| *a n d Digit*
| *a n d Letter*
| *a n e* | *a n f* | ... | *a n z*
| *a o* | *a p* | *a q*
| *a r Digit*
| *a r a* | *a r b* | *a r c* | ... | *a r q*
| *a r r Digit*
. . .

a → "a" | "A"
b → "b" | "B"
c → "c" | "C"
. . .

This description of *Identifier* works by describing all initial prefixes that could not start a keyword, and

specifying that an *Identifier* can be any of these prefixes optionally followed by letters or digits. Thus any string starting with the letters *h*, *j*, *k*, *q*, *x*, *y*, and *z* must be an identifier, since no Pascal keyword starts with one of these letters. The letter *a* begins the keywords **and** and **array** so the rules above give the two letter strings starting with *a* that cannot start one of these keywords. Next it gives the three letter strings that cannot start a keyword, even though they start with two letter strings that could.

Continuing the description in the pattern given above to avoid generating all the reserved keywords of Pascal would lead to a grammar for *Identifier* with 2,400 rules. Perhaps shorter descriptions could be given by defining symbols that represent all letters but one, but although these descriptions may be shorter they would still require hundreds of rules, and they would not be any clearer than the above one. Furthermore, the method given above assumes some other method has been devised for resolving the longest-match ambiguity. If the longest-match ambiguity were also resolved by grammar rewriting, the grammar size would grow by another large factor.

Apart from being too long to be considered human readable, a grammar disambiguated as above would generate enormous parsers with thousands of states, if the ordinary LR parser construction algorithms are used. The grammar would also be quite unmaintainable, since the addition or deletion of a single reserved keyword from the grammar would require the modification of hundreds of rules.

Even when grammar rewriting does not have such drastic effects on the size of the grammar, it can seriously affect the clarity of the grammar. Consider for instance the common way of rewriting grammars to eliminate the dangling-else ambiguity for Pascal illustrated by the grammar fragments in Figures 2.3 and 2.4.

Statement → *Labels ULStatement*

Labels → ϵ
| *Integer* ":" *Labels*

ULStatement → ϵ
| *Id* ":=" *Expr*
| *VarExpr* ":=" *Expr*
| *Id*
| *Id* "(" *ExprList* ")"
| **BEGIN** *StatList* **END**
| **IF** *Expr* **THEN** *Labels ULStatement*
ELSE *Labels ULStatement*
| **CASE** *Expr* **OF** *CaseBody* **END**
| **WHILE** *Expr* **DO** *Labels ULStatement*
| **REPEAT** *StatList* **UNTIL** *Expr*
| **FOR** *Id* ":=" *Expr UpDown Expr*
DO *Labels ULStatement*
| **WITH** *Var* **DO** *Labels ULStatement*
| **GOTO** *Integer*

Figure 2.3: A fragment of a Pascal grammar that exhibits the dangling-else ambiguity.

Statement → *Labels Matched*
| *Labels UnMatched*

Labels → ϵ
| *Integer* ":" *Labels*

Matched → ϵ
| *Id* ":=" *Expr*
| *VarExpr* ":=" *Expr*
| *Id*
| *Id* "(" *ExprList* ")"
| **BEGIN** *StatList* **END**
| **IF** *Expr* **THEN** *Labels Matched* **ELSE** *Labels Matched*
| **CASE** *Expr* **OF** *CaseBody* **END**
| **WHILE** *Expr* **DO** *Labels Matched*
| **REPEAT** *StatList* **UNTIL** *Expr*
| **FOR** *Id* ":=" *Expr UpDown Expr* **DO** *Labels Matched*
| **WITH** *Var* **DO** *Labels Matched*
| **GOTO** *Integer*

UnMatched → **IF** *Expr* **THEN** *Statement*
| **IF** *Expr* **THEN** *Labels Matched* **ELSE** *Labels UnMatched*
| **WHILE** *Expr* **DO** *Labels UnMatched*
| **FOR** *Id* ":=" *Expr UpDown Expr* **DO** *Labels UnMatched*
| **WITH** *Var* **DO** *Labels UnMatched*

Figure 2.4: The traditional method of rewriting a Pascal grammar to resolve the dangling-else ambiguity.

The grammar fragment of Figure 2.4 is only slightly longer than the equivalent ambiguous grammar fragment of Figure 2.3. The real disadvantage of this rewriting is that it requires the duplication, with only slight modification, of a number of productions, i.e. close coupling of grammar rules. Duplicating productions has the same bad effect on a grammar that duplicating code has on a program: it increases the chance of error during maintenance in that changes to one production must be carefully checked to see if the accompanying duplicate production also needs to be changed. In addition the differences between the duplicated productions in this example are subtle and require careful attention.

2.4.2. Disambiguation by Default Action

Wharton[57] proposes a set of rules that can be used to order all possible parses of ambiguous sentences. He then shows how bottom-up parsers and top-down parsers can be made to generate only the first of these parses. In other words the disambiguation of the grammar is implicit in the form and ordering of the grammar rules. Methods similar to his are in actual use by existing translator-generating tools, but they lead to a practical problem: Should the ambiguity be reported or not? A "no" answer leads to a dangerous translator, and a "yes" answer leads to an unusable translator.

The scanner generator LEX[38] has implied rules for dealing with ambiguities: the longest match and the earliest rule take precedence, in that order. This means that no ambiguities are reported to the user, neither the intentional ones nor the unintentional ones. This strategy is not acceptable in general, since unintentional ambiguities could only be uncovered by testing the generated scanner, and generating a validation suite for identifying all possible errors may be difficult or impossible. On the other hand, reporting all of the conflicts and how they were resolved, as YACC does, would generate tens or hundreds of messages that must each be carefully checked for correctness.

2.4.3. Disambiguation by a Screener Phase

DeRemer,[14] in his multiphase, single metalanguage parser generation system, uses a screener phase to resolve the reserved-keyword ambiguity. The screener tests each identifier recognized by the scanner to see if it is actually a reserved keyword. He does not give a notation for describing the screener phase, but presumably this would require a metalanguage distinct from the other two phases. Nor does he present a solution for the longest-match ambiguity. Since he uses an LR(1) parser for the scanner phase, his system could resolve all longest match ambiguities by selecting the **shift** action in all shift-reduce conflicts. This conflict resolution method would have the same drawbacks of the approach used by LEX, in that unintentional conflicts would be resolved without being reported as well as the intentional ones.

2.4.4. Explicit Disambiguation Rules

Aho, Johnson, and Ullman[3] discuss parser generation for grammars with disambiguation rules. They argue that an ambiguous grammar can be clearer and more concise than an unambiguous grammar for the same language, and can lead to smaller faster parsers. They propose two types of highly-specialized disambiguation rule. Their first type of rule is designed specifically to simplify the specification of operator precedence in a programming language. Their second kind of rule applies specifically to the dangling-else problem. The first type of rule is available with the YACC parser generator, and has successfully been used in generating parsers.

Aho et al. do not give a formal description of the semantics of their disambiguation rules, nor attempt to classify the family of languages describable with their notation. Their rules are defined only in terms of how they affect LR parser generation from a grammar containing them. Furthermore their two rules are inadequate for resolving the common ambiguities in character-level grammars in any reasonable way.

2.4.5. The Proposed Disambiguation Method

The proposed method of disambiguating programming-language grammars is to include explicit disambiguation rules in the grammar, as is proposed by Aho, Johnson, and Ullman,[3] but to use disambiguation rules with a more general form. The principle of the proposed disambiguation rules is that whenever the grammar allows more than one derivation of a sentence, the disambiguation rules provide restrictive information that forbids all but one of those derivations. A CFG that includes one or more of these restrictive rules is called a restricted CFG (RCFG).

We propose two restrictive rules, the *exclusion* rule, and the *adjacency-restriction* rule. The exclusion rule takes the form

$$A \nrightarrow \alpha.$$

It specifies that despite the other grammar rules, the symbol A may not generate sentences in the language $L(\alpha)$. This rule has obvious applications in inhibiting the generation of reserved keywords by the identifier symbol.

The adjacency-restriction rule takes the form

$$W \not\dashv X.$$

When added to a grammar G, it specifies that sentences in $L(G)$ may not contain a sentence generated by W immediately followed by a sentence generated by X. This rule can be used to resolve

longest match ambiguities, including the dangling-else ambiguity.

A full description of these rules is given in Chapter 5. They are defined by giving formal descriptions of how they affect derivations. Grammar transformations are also presented that convert an RCFG containing the new disambiguation rules into a pure CFG that generates the same language. Since the transformations tend to generate very large grammars that are unsuitable for parser generation, parser generation methods that accept untransformed RCFG's are presented. The parser generation methods accept only a restricted use of the disambiguation rules, but these usages are adequate for describing modern programming languages.

Although these restrictive rules are proposed for disambiguating grammars, that is not their only possible use. They can also be used to place restrictions on an unambiguous grammar. Such uses sometimes lead to shorter, clearer grammars.

Chapter 3

Using Existing Tools to Implement Single-Metalanguage Translators

This chapter presents two attempts to use existing parser generation tools to implement single-metalanguage recognizers for the programming language Pascal. The first recognizer, called *PRYY*, has two phases, a scanner phase and a parser phase, but both phases are described by the same metalanguage, BNF. The second recognizer, called *PROPY*, attempts to implement the Pascal recognizer in a single phase using BNF. The object is to illustrate the problems that one encounters when trying to give a full description of a programming language in a consistent metalanguage without the use of enhanced parser lookahead, or special disambiguation rules. Pascal was chosen as the test language because it is a reasonably modern language, but at the same time represents a minimal test case. If the approach does not work well for Pascal, then it cannot be expected to work well for larger languages such as Ada.

The UNIX utilities YACC[27] and LEX[38] were chosen as the parser implementation tools. YACC is representative of a standard LALR parser generator and has been in use long enough to be stable and reliable. LEX is used as an FSM scanner generator when a traditional scanner is needed for comparison purposes. LEX is also stable and reliable, but is not as widely used as YACC because hand-coded scanners can be significantly more efficient than automatically generated ones.

3.1. PRLY: The Standard Recognizer

A Pascal recognizer implemented using traditional techniques was used as a standard against which to compare the proposed recognizers. The standard recognizer, called *PRLY*‡ has a finite-state scanner generated using LEX and a DPDA parser generated using YACC. Appendix A presents the grammars for the scanner and the parser.

3.1.1. The Grammar for the Standard Scanner

The grammar for the standard scanner consists of a list of patterns, each one a regular expression. An input string is processed from left to right by attempting to match a prefix of the unprocessed input with one of the patterns. Each time a match is found, the corresponding scanner action is executed, as specified by the C-language program fragment to the right of the pattern.

The grammar may be ambiguous in that it may be possible for more than one of the patterns to match parts of the same prefix. In this case LEX uses two disambiguation rules:

1) The pattern matching the longest prefix of the unprocessed input is preferred.
2) If two or more patterns match the same length prefix then the pattern appearing earliest in the grammar is preferred.

In this way all ambiguities, intentional and unintentional, are resolved.

The grammar presented uses two special features of LEX: *substitution strings* and *start states*. Substitution strings allow names to be given for frequently occurring subpatterns. The subpatterns are be substituted in place wherever their names are used. In the grammar in Appendix A, substitution

‡ The names PRLY, PRYY, and PROPY can be explained as follows: PRLY is a Pascal recognizer with a LEX written scanner and a YACC written parser, PRYY is a Pascal recognizer with a YACC written scanner and a YACC written parser, and PROPY is a Pascal recognizer with only one phase written by YACC.

strings are defined for the symbols *letter*, which represents any letter of the alphabet, *digit*, which represents any digit from 0 to 9, and each letter of the alphabet, which represents either the upper- or lower-case version of the letter. This device saves pattern repetition and thus shortens and simplifies the grammar.

The other special feature, start states, provides the grammar writer with explicit control over the finite state machine generated. In the standard LEX model every pattern is equally likely at the start of pattern matching, but with start states a grammar writer can specify specific recognizer states from which only certain patterns are active. This often allows a simpler grammar, but in our grammar start states are used to assist in the recognition of comments. Start states are a standard method of getting LEX to recognize unlimited length comments. Normally the string that LEX matches with a single pattern must fall entirely inside its input buffer. This is necessary so that LEX can match a shorter pattern should a long match fail. Since Pascal comments have no length limit, only a buffer as long as the longest expected source file would suffice, which is not always practical.

3.1.2. The Grammar for the Standard Parser

The grammar for the standard parser is very similar to the grammar that appears in the ISO Pascal definition.[26] The principal difference is that some nonterminals have been made into terminals. These include the nonterminals Identifier, Label, UnsignedInteger, UnsignedReal, and CharacterString. All terminals and nonterminals that were used solely to define these new terminals have been removed from the grammar.

In addition all multicharacter terminals such as the reserved keywords and multicharacter operators, have been replaced by symbol names. For instance the string "begin" was replace by the terminal symbol BEGIN, and the string "<=" was replaced by the symbol LE. It is the scanner's job to recognize these multicharacter terminals and pass their symbol names to the parser.

Another difference of the presented grammar from the ISO standard grammar is that the dangling-else ambiguity has been resolved by rewriting the grammar.

3.1.3. The Interface Between the Standard Scanner and Parser

The grammar for the parser contains a series of %token statements and a %union statement that partly define the interface between the scanner and the parser. The %token statements give symbolic names to each of the tokens that can be passed from the scanner to the parser, and may also give (in angle brackets) the type of the token value to be passed. The %union statement gives symbolic names to the elements of a C-language type union that contains all possible types that a token value returned by the scanner can take. The type of the value associated with a token is specified by using a symbolic name from this union.

One can consider the %token and %union statements of YACC, along with the C-language type declarations as a two-language system for describing the scanner-parser interface. Alternatively one can think of the C-language file generated by these statements as a single-language description of the interface. Perhaps the second view is better because the parser writer usually must augment the YACC produced description of the interface with his own hand-written C code. In the case of the standard recognizer, the C-language file "extern.h," shown in Appendix A, represents just such an augmentation.

The completed description of the interface between the standard scanner and the standard parser is given by the two include files "tokens.h" and "extern.h" shown in Appendix A. The file tokens.h contains C-language preprocessor definitions establishing integer values for the symbolic names of the tokens. This mechanism establishes the link between symbolic names used for tokens in the parser and the scanner. Tokens.h also contains type definitions for token values transmitted from the scanner to the parser.

The file extern.h is a manually prepared description of the shared variables that communicate execution time parsing information between the two phases. In this case three variables are communicated: the current line number in the input file (`yylineno`), the characters comprising the current

token (`yytext`), and the value of the current token (`yylval`). The first two variables contain information useful when reporting parsing errors. If a true Pascal parser were being presented, rather than simply a Pascal recognizer, then the third variable `yylval` would also be useful for communicating the value of the current token. The value of the current token would be such things as the value of numeric and string constants, and the characters comprising an identifier.

Compulsory Compilation-Order of the Modules

In addition to the fact that the interface between the parser and the scanner needs a language (or two) for its description, it imposes an ordering on the compilation of the modules of the recognizer. A desirable feature of the modularization of a program is that any particular module need only be recompiled if the source for that module changes or if the interface with the other modules changes. Since the interface description file is produced by the parser generator, the parser must be generated before the scanner. The problem is that the parser generator produces an interface description every time it is run, whether or not the interface has changed since the last parser generation. In the simplest scheduling of the generation of the recognizer, the scanner would have to be recompiled whenever either the parser or the scanner changes, thus negating some of the benefits of modularization.

Scanner generation often requires significant processor time, and except on very fast machines, significant real elapsed time. As a result, the simplest scheduling of the scanner generation is often undesirable. The alternative is to always keep the previous description of the interface so that a comparison with the new description can be automatically performed, and scanner generation done only if the interface has changed or the scanner description has changed since the last generation.

The makefile included in Appendix A reflects all these timing constraints imposed on the generation of the standard recognizer. The parser grammar must be checked for changes before the scanner grammar. If the parser grammar has changed, both the parser and the interface description are regenerated. Then the new interface description is compared to the old one. If the interface description or the scanner grammar has changed since the last scanner generation, a new scanner is generated. The other part of the interface description "extern.h" can be treated as any other source file since it is generated by hand.

3.2. PRYY: A Two-Phase Single-Metalanguage Pascal Recognizer

A standard result of formal language theory is that the languages describable by regular expressions form a proper subset of the languages describable by context free grammars. This leads to the observation that the complexity of preparing and processing the recognizer presented above could be reduced by describing the scanner with a context-free grammar rather than with regular expressions. The complexity of preparing the recognizer would be reduced since it would mean that the same programming tool, YACC, could be used to generate both phases of the recognizer. This change would in turn mean that the grammar writer would need to consult only one reference manual when preparing the two grammars.

To test this approach we designed and tested a two-phase single-metalanguage Pascal Recognizer. The recognizer is two-phase in that it still has a scanner phase and a parser phase, but both phases are described using a context-free grammar.

3.2.1. The Grammar for YaccScn

Appendix B presents a context-free grammar for a scanner called *YaccScn*. YaccScn is designed to replace the scanner in the standard Pascal recognizer. An examination of the grammar shows that it has a great deal of similarity to the grammar for the standard scanner, even though they are written in different metalanguages. Each regular expression for a token recognized by the standard scanner has at least one rule in the context-free grammar for YaccScn.

Each substitution string, of the grammar for the standard scanner, is also replaced by context-free productions. Thus there are productions for the nonterminals *letter* and *digit,* as well as productions for each letter of the alphabet to get case-insensitive symbols *a* to *z* representing each letter.

This change is gratifying, since it represents a replacement of a LEX extension to regular expression notation, by ordinary rules of a context-free grammar.

3.2.1.1. Resolving Ambiguities in YaccScn

LEX uses the simple scheme described above, in Section 3.1.1, for resolving ambiguities in input grammars. How are ambiguities resolved in the grammar for YaccScn? YACC has disambiguation rules similar to those of LEX. All ambiguities in a grammar input to YACC result in shift-reduce or reduce-reduce conflicts during the generation of the LALR(1) parse tables. (Grammars that are not ambiguous may also generate such conflicts if they are not LALR(1).)

When YACC encounters a shift-reduce conflict, it resolves the conflict in favour of a **shift** action. This disambiguation rule has a similar effect to the longest match rule of LEX. Rather than reducing a string to some nonterminal, Yacc shifts more symbols onto the parsing stack in the hopes of reducing a longer string.

When YACC encounters a reduce-reduce conflict, in which any of two or more rules can be used to reduce a string of stack symbols, YACC resolves the conflict in favour of the earliest rule in the grammar. This disambiguation strategy has an effect similar to LEX's strategy of using the earliest pattern in the grammar when two or more patterns match the same input prefix.

The main difference between the strategies of LEX and YACC for resolving ambiguities is that YACC reports all ambiguities no matter how they are resolved, whereas LEX reports none. This means that all unintentional ambiguities are reported, but it also means that the grammar writer must examine dozens or hundreds of messages about intentional ambiguities to find any about unintentional ones. The grammar for YaccScn, for instance, generates 265 intentional shift-reduce conflicts.

The grammar for YaccScn is a large segment of a grammar for Pascal, and as such it contains many of the ambiguities and deficiencies common to Pascal grammars as listed in Section 1.2. In particular it has the problems of distinguishing reserved words from identifiers, and recognizing the longest match possible for identifiers and numeric constants. In addition, since the grammar for a scanner is a subset of a full Pascal grammar, it has many additional ambiguities that are normally resolved by the full grammar. For instance the strings ":", "=", and ":=" are all valid Pascal tokens. As a result, a scanner does not know whether to report the string ":=" as two tokens or one. The full grammar for Pascal, however, does not permit a colon token to be followed by an equal-sign token; the two must be recognized as an assignment token when appearing together. Similar problems occur with the strings "(.", ".)", "..", "<=", ">=", and "<>". In addition, the scanner grammar does not have the information contained in a full Pascal grammar, that an identifier never immediately follows another identifier, or a numeric constant never immediately follows another numeric constant. Thus a naive scanner grammar could equally well break up a multiple-character identifier or numeric constant, into many identifiers or numeric constants. Fortunately all these ambiguities are correctly resolved by favouring a **shift** action over a **reduce** action in a shift-reduce conflict, and thus constitute some of the ambiguities that were intentionally left in the grammar.

Reserved-Keyword Ambiguities

In the parser generated for YaccScn, a conflict arises as soon as the first letter of a keyword appears on the parsing stack. Identifiers are described as sequences of the nonterminals *letter* or *digit*, whereas keywords are described as sequences of individual letters themselves. Thus after pushing the first letter of a keyword onto the parsing stack the parser must decide whether to reduce that letter to the symbol *letter*, or leave it unreduced and push on the next letter hoping that a complete keyword is forthcoming. YACC resolves this conflict by shifting, which means that there must be some way to correct this decision should it turn out that the remainder of the input did not in fact provide a complete keyword. To do this, the grammar for YaccScn provides the symbol *partial_keyword* that matches all possible prefixes of the Pascal reserved keywords. In this way, if the decision to recognize a keyword was the wrong one, the parser can correct the error by recognizing a partial keyword instead, and reduce this partial keyword to an identifier. If a full keyword is recognized, no reduce-reduce conflict arises because the grammar does not provide a way to reduce full keywords to

identifiers, only partial keywords, or full keywords followed by an alpha-numeric character. This technique has added considerably to the length of the grammar, and to the difficulty of maintaining the grammar to reflect language modifications should any arise.

Transferring Information from the Complete Grammar

Some information about the complete grammar for Pascal has been artificially injected into the grammar for YaccScn. The grammar for YaccScn actually recognizes a language consisting of a stream of Pascal tokens (more on this in the next section). The parser recognizes one token at a time, and iteratively combines it with the token stream recognized so far, allowing optional white space to appear between the tokens. The grammar, however, remembers the kind of the last token recognized and does not allow certain kinds of tokens to be immediately followed by other tokens. Using the nonterminals *an_token_stream*, *num_token_stream*, and *token_stream,* the grammar remembers three kinds of previous tokens, alpha-numeric tokens (such as keywords and identifiers), numeric tokens (such as integer and real constants), and any other type of token. The grammar specifies that an alpha-numeric token cannot be followed by another alpha-numeric token or by a numeric token, unless they are separated by actual (not optional) white space. If an alpha-numeric token were immediately followed by another alpha-numeric token, then the two should have been combined into a single longer identifier token. The grammar applies a similar restriction to numeric tokens. This strategy is used to significantly reduce the total number of shift-reduce conflicts generated by the parse-table generator, thus reducing the effort of checking all conflicts for errors.

3.2.1.2. Deficiencies of the Grammar for YaccScn

The grammar for YaccScn presented in Appendix B is not correct for ISO Pascal; it has two deficiencies. The first deficiency is that the grammar for YaccScn accepts only braces, "`{`" and "`}`", as comment delimiters, whereas ISO Pascal allows the alternate delimiters "`(*`" and "`*)`". The problem is that comments can appear anywhere that white space is permitted, and that the open parenthesis is also a common Pascal delimiter, thus many shift reduce conflicts are generated. Since white space can occur in so many places, it is not reasonable to factor out the recognition of "`(*`" as was done for keywords.

Accepting only one form of comment delimiter simplifies the grammar for YaccScn, and hence may give it an unfair advantage in a comparison with a LEX generated scanner. To eliminate this possibility, the LEX scanner grammar for the standard recognizer was rewritten, and simplified, so that it too accepts only braces as a comment delimiter.

The second deficiency of the grammar for YaccScn is that it has an ambiguity between the descriptions of real constants and of integer constants that precede a range designator "`..`". Thus the Pascal range type "`1..10`" will be rejected. The problem is that after reading the first integer constant and looking ahead at the dot, the scanner must decide whether to recognize an integer constant and pass it on to the parser, or to shift on the dot hoping for a real constant. YACC resolves the conflict by shifting, which is the incorrect action if it really is a range constant. To use range constants with this recognizer, the first integer constant must be separated from the dot-dot range token by real white space. So the above type would have to be written "`1 ..10`". The problem does not occur if the first constant of the range specifier is a symbolic constant.

This deficiency is very hard to correct in a two-phase grammar since floating-point numbers are recognized in one grammar and ranges in another. A great deal of effort did not yield a suitable partitioning of the grammar and the search was abandoned. Unlike the deficiency with comment delimiters, however, the LEX generated scanner was not rewritten to also contain this defect. Changing the LEX grammar so that it too would contain the defect would actually slow down the standard recognizer, thus giving it an unfair disadvantage due to a problem it did not create.

In the comparisons of PRLY and PRYY that follow, remember that the rules of Pascal had to be bent to accommodate the single-metalanguage version, and hence the comparison is not absolutely fair. Nevertheless the version of Pascal accepted by the single-metalanguage version is close enough to ISO Pascal to yield useful comparison information.

3.2.2. Technical Implementation Problems

Unlike LEX, YACC was designed for writing parsers not scanners. YACC is designed to be invoked, recognize one and only one sentence in a language, and then return to the calling program. LEX on the other hand can be thought of as recognizing a sequence of sentences of a language, and returning to the calling program after each sentence. Thus a LEX written scanner can be called as a coroutine by a YACC written parser to fetch tokens one after another. A YACC written scanner, however, does not return until it has verified that there is no further input beyond exactly one sentence of the given language.

To circumvent this shortcoming, the YACC written scanner and the YACC written parser are run sequentially instead of as coroutines. The scanner fills a large memory array with the integer codes for the tokens recognized and the parser reads this array as input.

3.3. Comparison of PRYY with PRLY

We conclude that except for the problems with comments and constant ranges, a context-free grammar is a reasonable way to describe a scanner for Pascal. Assuming that a customized parser generator could provide convenient ways to solve these two problems, the main concerns remaining would be about the size and speed of the generated scanners. Table 3.1 summarizes the characteristics of PRLY and PRYY.

Name	Phase	Rules	States	Object Size (Bytes)	Parse Time† (Sec.)
PRLY	scanner	97	545	21.9k	
	parser	204	369	6.8k	
	total	301	914	28.7k	49.7
PRYY	scanner	411	442	11.1k	
	parser	204	369	6.8k	
	total	615	811	18.9k	67.9

Table 3.1: A comparison of size and speed of PRLY versus PRYY.

The first thing to notice in Table 3.1 is that the number of rules in the grammar for the LEX generated scanner is about one quarter of the number for the YACC generator scanner. A significant contributing factor to this discrepancy is the large number of rules needed to describe the nonterminal *partial_keyword,* and this is a notable deficiency of the grammar. Nevertheless the rule count for the grammar for YaccScn is artificially exaggerated by counting each alternate definition for a nonterminal as a separate rule. Thus the definition of the nonterminal *letter* is counted as 26 rules in the grammar for YaccScn, whereas the specialized notation of LEX allows it to be counted as one rule in the grammar for the standard scanner. Thus a customization of YACC's notation could lead to a lower rule count for YaccScn than shown in the table.

The next thing to notice is that despite the higher rule count of the grammar for YaccScn, the number of states in the scanner for PRYY, as reported by YACC, is only 89% of the number of states in the scanner for PRLY, as reported by LEX. Similarly the size of the object code for the scanner for PRYY is 69% of the size of the scanner for PRLY. These ratios are a further indication that the rule count for PRYY's scanner was artificially inflated, and it alleviates fears that a DPDA scanner will be too large.

† Execution time required to parse 8,564 lines of Pascal code on a MicroVAX II.

Even a comparison of the running times of the two recognizers is encouraging. Although PRYY had a running time about 1.5 times that of PRLY, this is acceptable performance, especially considering that YACC and its input language are being used for a purpose for which they were not designed.

These result shows that the size and efficiency of a dual-phase single-metalanguage Pascal recognizer can be comparable to those of a traditional dual-metalanguage one.

3.4. PROPY: A Single-Phase, Single-Metalanguage Pascal Recognizer

The experiments with the two-phase single-metalanguage Pascal recognizer PRYY were successful enough to consider an even simpler strategy. Why not combine the context-free grammars for the scanner and the parser into a single grammar? Such a strategy would eliminate the interface between the two phases, since the names of the nonterminals representing tokens in the scanner grammar, match the names of the terminals in the parser grammar. In addition, the combining of the two grammars would eliminate the part of the grammar for YaccScn that describes the three token-stream nonterminals. The token-stream nonterminals were introduced primarily to transfer a limited amount of information about legal token sequences from the parser grammar to the scanner grammar. By combining the two grammars these rules would become unnecessary.

To test the idea of a combined grammar, the grammar for YaccScn was combined with the parser grammar, and the resulting grammar is shown in Appendix C. The grammar was prepared to be used as input to YACC.

3.4.1. Conflicts Resolved by the Combined Grammar

Several conflicts in the grammar for YaccScn are resolved by combining it with the grammar for the parser. As was mentioned in Section 3.2.1.1, "Resolving Ambiguities in YaccScn," the grammar for YaccScn could permit the multiple character tokens ":=", "(.", ".)", "..", "<=", ">=", and "<>". to be parsed as one token or two. The grammar for the parser, however, indicates that in none of these cases can the two component tokens appear in sequence in a valid Pascal program. The combined grammar also contains this information, and hence conflicts on the parsing of these multiple character tokens disappear.

Similarly, part of the purpose of the special nonterminals *an_token_stream* and *num_token_stream* in the grammar for YaccScn was to indicate that in a Pascal program an identifier never immediately follows another identifier, and a numeric constant never immediately follows another numeric constant. The parser grammar contained this information, so the combined grammar also contains it.

3.4.2. Conflicts Inherited from the Two-Phase Grammar

Most of the conflicts in the grammar for YaccScn were not resolved by combining it with the grammar for the parser; they were simply passed on. These conflicts include the ones due to the ambiguity between reserved keywords and identifiers. Since it is possible for keywords and identifiers to follow each other in Pascal, it is still necessary to include the numerous rules for the symbol *partial_keyword* in order to be able to recover from erroneous parsing decisions.

3.4.3. New Problems and Conflicts in the Combined Grammar

Optional White Space

In the grammar for YaccScn, all usage of the nonterminal "optional_white" was confined to the description of token streams. The combined grammar no longer has such a convenient way to localize the use of white space. Instead, the combined grammar explicitly specifies that optional white space may follow the recognition of any nonterminal that formerly represented a token. Thus such symbols as "ASSIGN_" and "PROCEDURE_", now have twin symbols without the underscores that represent the same strings as the old symbols, but with optional white space appended. This approach is simple but gives a somewhat bulky grammar.

An alternate approach to handling white space would be to distribute the white space between the symbols of the rules that were formerly parser rules. This approach introduces reduce-reduce conflicts not present in the former approach. These new conflicts appear for each rule that has a nullable symbol between two white-space symbols, since there are then three consecutive symbols that can be reduced from ϵ. Nevertheless, if white space is represented by some non-obtrusive symbol such as "_", some grammar writers may find a grammar written according to the latter approach more aesthetically pleasing, since it more accurately represents the meaning of optional white space.

Required White Space

The next significant problem with the combined grammar is that when the token-stream symbols are removed, the only convenient way to specify that reserved words and identifiers must be separated by real (not optional) white space, is also lost. The problem is that there are many high-level constructs such as expressions and symbol lists that can be comprised of a single identifier, and may occur immediately preceding or following a keyword. To express the requirement that keywords and identifiers must be separated by real white space, duplicate symbols would be required for all high-level constructs to represent when those constructs generate sentential forms that can begin with an identifier or end with an identifier. Then many grammar rules would have to be duplicated and modified to specify when real white space is required. This grammar rewriting would lead to an explosion in the size and complexity of the grammar.

Reduced Parser Lookahead

Perhaps the biggest problem introduced by combining the two grammars is the loss of lookahead power. Formerly the parser generated by YACC could look ahead at an entire token to decide the next parser action. In the combined parser grammar, the parser can look ahead only a single input character. In the grammar for Pascal, there are several constructs that cannot be parsed (without unwieldy grammar rewriting) as a result of this problem. A comprehensive list of these problems follows.

1) When parsing a compound statement such as

```
BEGIN
    Statement_List
END
```

if the last statement of the statement list ends in a semicolon, as is permissible, then the parser cannot decide whether the "E" of `END`, is the beginning of a new assignment statement, or is the start of the END keyword. The decision affects whether the parser reduces the statements read so far, or shifts on more input hoping for a longer statement list. YACC's default preference for shift means that ending a statement list inside a compound statement with a semicolon will always cause a parsing error. A similar problem arises with the reserved word "`UNTIL`" in the REPEAT-UNTIL construct, and in distinguishing the reserved word "`ELSE`" in an IF-THEN-ELSE construct from the reserved word "`END`".

2) A CASE construct takes the form:

```
CASE Expression OF
    Case_List
END
```

After shifting on the characters that comprise the case expression, the parser cannot decide whether the lookahead symbol "`O`" is the start of the keyword `OF` or of the keyword `OR`. In the former case it should reduce the symbols to an expression, whereas in the later case it should continue shifting terminals in the hope of reducing a longer string to an expression. YACC's default action of **shift** means that a CASE construct can never be parsed properly.

3) A WHILE loop takes the form:

```
WHILE Expression DO
    Statement;
```

After shifting on the characters that comprise the while expression, the parser cannot decide whether the lookahead character "D" is the start of the keyword DO or of the keyword DIV. In the former case it should reduce the symbols to an expression, whereas in the later case it should continue shifting terminals in the hope of reducing a longer string to an expression. YACC's default action of **shift** means that a WHILE construct can never be parsed properly. The same problem occurs with the FOR construct.

4) After shifting on the characters of a variable, if the lookahead character is a dot, the parser cannot decide whether the dot starts the sequence ".)" and the variable should be reduced to an expression, or whether the dot represents the dot of a record-field selector, and it should be shifted on to be reduced as part of a longer variable string. YACC's default action of **shift** means that the sequence ".)" can never be used as the alternate for "]".

5) After recognizing the characters that comprise a CONST, TYPE, or VAR declaration section, with one or more of the lookahead characters "T" (for TYPE), "V" (for VAR), "P" (for PROCEDURE), "F" (for FUNCTION), or "B" (for BEGIN), the parser cannot decide whether to terminate the recognition of the current declaration section, or shift on the lookahead character in the expectation that it is part of a variable name introducing a new variable declaration in the current section. YACC's default action of **shift**, implies that no declarations section will ever be completed, and hence YACC reports those rules as never being reduced.

6) In ISO Pascal it is legal to specify an empty record declaration. Thus the string RECORD END specifies a legal type declaration. Using the combined grammar, however, YACC cannot decide on the basis of the lookahead character "E" whether to reduce the field list from ϵ, or whether to shift on the letter "E" in the expectation that it is part of a variable name that initiates a record field declaration.

7) In a procedure or function declaration the body of the procedure may be replaced by special identifiers such as FORWARD or EXTERNAL. After shifting on the characters comprising a procedure header, and looking ahead at the characters "L" (for LABEL), "T" (for TYPE), "V" (for VAR), "P" (for PROCEDURE), "F" (for FUNCTION), or "B" (for BEGIN), YACC cannot decide whether these letters start declarations belonging to the procedure body, or whether they initiate an identifier that replaces the procedure body. The combined grammar, and YACC's default action of **shift**, means that the parser will never initiate the recognition of declaration, at the start of a procedure or function body.

8) After encountering an identifier and looking ahead at the character "(" the parser cannot decide whether to reduce the identifier to a procedure or function invocation header, or shift on the parenthesis in the expectation that it comprises the first character of the sequence "(." as a substitute for "[" which would indicate a subscripted variable. YACC's default action blocks the recognition of procedure or function invocations.

3.5. Conclusions

The experimental Pascal recognizer PRYY shows that it is possible to describe a reasonable scanner using an LALR(1) grammar and simple conflict resolution. The remaining shortcomings, such as the complex description of identifiers, the unimplemented Pascal characteristics, and the large number of defaulted conflicts remaining are serious problems but not insurmountable. Most importantly PRYY shows that a DPDA scanner can compete in efficiency with an FSM scanner, both in size and speed.

The experiments with PRYY and PROPY show that to be able to describe a recognizer briefly and conveniently, the metalanguage should be enhanced. The metalanguage needs a method for concisely disambiguating simple constructs, since the only available method—adding more CF rules—is not powerful enough.

The experiment with PROPY also shows that a one-phase recognizer with a reasonable grammar needs a more powerful lookahead capability than the single character lookahead provided by an LALR(1) parser.

Chapter 4

Improved NSLR(1) Parser Generation

4.1. Introduction

In the previous chapters, the inadequacies of LR(1) parsers for implementing single-phase translators were presented, and in Chapter 2 the reasons for choosing noncanonical SLR(1) parsers, as presented by K. C. Tai,[55] to overcome those inadequacies were given. This chapter presents a detailed description of NSLR(1) parsers and parser generators. There are, however, some deficiencies in the parser generation algorithm as presented by Tai, and so we also propose methods of correcting those deficiencies.

The first deficiency treated here is in 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. We propose a method for reducing the number of ϵ-reducing items generated, and in this way admit a class of useful grammars previously rejected. Correcting this deficiency permits the correction of a second deficiency. There is an error in Tai's algorithm that may omit necessary nonterminals from the lookahead sets of complete items in ordinary states, if the parser also contains noncanonically expanded states. This error will be demonstrated and a correction proposed. Unfortunately, the correction proposed for this error generates lookahead sets with many useless entries. Hence we also propose methods for reducing the number of these useless entries.

4.2. Standard Parsing Terminology

The notation and terminology used in this chapter and the remainder of this thesis is largely standard and can be found in such sources as Aho, Sethi and Ullman,[5] Hopcroft and Ullman,[24] Harrison,[22] and Aho and Ullman.[1] In particular, with very few exceptions, it is the same terminology used by Tai.[55] To speed the reading of this section by experienced readers, any differences from standard notation have been marked by a filled triangle (▸), and differences from Tai's notation have been marked by a filled box (■).

A *context-free grammar* (CFG) G is a quadruple $G = (V_N, V_T, P, S)$, where ▸ V_N is the set of *nonterminals*, ▸ V_T is the set of *terminals*, P is the set of *productions*, and S in V_N is the *start symbol*. The set of all grammar symbols is represented by $V = V_N \cup V_T$. We assume that the grammar has no duplicate or useless productions, and no useless symbols.

Lowercase letters early in the alphabet, such as a, b, and c, represent a single terminal symbol. Uppercase letters early in the alphabet, such as A, B, and C, and also the letter S, represent nonterminals. Uppercase letters late in the alphabet, such as X, Y, and Z, represent terminals or nonterminals. The Greek letter ϵ represents the empty string, and the other Greek letters, such as α, β, and γ, represent strings of terminals or nonterminals, or ϵ. (■ Tai used lowercase letters late in the alphabet for this purpose.) Lowercase letters late in the alphabet represent strings of terminals or ϵ. The length of a string α is denoted by $|\alpha|$, and therefore $|\epsilon| = 0$. Letters representing strings of symbols can be subscripted, so that α_i represents the i^{th} symbol of α. The symbol $\varnothing$ represents the empty set, (■ Tai used the Greek letter ϕ for the empty set.)

The productions in P are numbered $1, 2, \ldots, p$ where $p = |P|$. Productions take the form $A \rightarrow \alpha$, where A is called the *left part*, and α is called the *right part*. Let P_i denote the i^{th} production in P, and ▸ let n_i denote the length of the right part of P_i.

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 $\beta A \gamma$ *derives* $\beta \alpha \gamma$. The transitive closure of $\Rightarrow$ is denoted by $\overset{+}{\Rightarrow}$, and the reflexive transitive closure of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$. There are two special kinds of derivation: a leftmost derivation denoted by $\alpha \underset{lm}{\Rightarrow} \beta$, and a rightmost derivation denoted by $\alpha \underset{rm}{\Rightarrow} \beta$. In the leftmost derivation $\alpha \underset{lm}{\Rightarrow} \beta$, the leftmost nonterminal in α is substituted to derive β, and in the rightmost derivation $\alpha \underset{rm}{\Rightarrow} \beta$, the rightmost nonterminal of alpha is substituted.

Every derivation with a grammar has a corresponding parse tree. In such a derivation, every application of a production $A \rightarrow \alpha$ corresponds to a node in the tree labeled A with $|\alpha|$ ordered descendants. ▸ We call two derivations by the same grammar *isomorphic* if they have the same corresponding parse tree.

A *sentential form* of G is a string α such that $S \overset{*}{\Rightarrow} \alpha$ 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., $L(G) = \{x \mid S \overset{*}{\Rightarrow} x\}$. We can also refer to the language generated by some symbol of grammar G as $L_G(X) = \{x \mid X \overset{*}{\Rightarrow} x\}$.

A symbol X in V is said to be *useless* if there is no derivation of the form $S \overset{*}{\Rightarrow} uXv \overset{*}{\Rightarrow} uxv$. A nonterminal A is called *nullable* if there exists a derivation $A \overset{*}{\Rightarrow} \epsilon$.

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

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

Definition: ▸ ■ Let T_FIRST(α) be the set of terminals in FIRST(α).

$$\text{T\_FIRST}(\alpha) = \{a \mid a \text{ is in } V_T, \text{ and } a \text{ is in FIRST}(\alpha)\}$$

Definition: Let LAST(α) be the set of symbols that can be the last symbol of any sentential form derivable from α.

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

4.3. A Review of Ordinary SLR(1) Parsing and Parser Generation

In order to simplify the description of noncanonical SLR(1) parsing and parsing, a review of ordinary SLR(1) parsing and parser generation is presented here.

SLR(1) grammars and parsers were first presented by deRemer.[13] An SLR(1) parser consists of a set Q of states and two functions, the *parser action function* f and the *goto function* g. State s_0 is designated as the initial state. Assume that each input string is preceded by the symbol $\vdash$ and followed by the symbol $\dashv$, where neither $\vdash$ nor $\dashv$ is in V_T. The parsing action function f takes an arbitrary state s and an input symbol b as arguments, where b is a terminal or in the set $\{\vdash, \dashv\}$. The value of $f(s, b)$ is either **shift**, **reduce** i, **error,** or **accept**. The goto function g takes a state s and a symbol Y in V as arguments, and the value of $g(s,Y)$ is either a state or **error.**

To construct a parser for a grammar $G = (V_N, V_T, P, S)$, G is augmented with a new start symbol S', not in V_N, and a new production $S' \rightarrow \vdash S \dashv$. The new production is assumed to be the zeroth production.

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

$$\text{T\_FOLLOW}(A) = \{b \text{ in } V_T \cup \{\dashv\} \mid S' \overset{*}{\Rightarrow} \beta A b \gamma \text{ for some } \beta \text{ in } \vdash V^* \text{ and } \gamma \text{ in } V^* \dashv\}.$$

Each state in an SLR(1) parser is represented by a set of *items* $[i, j]$ where i is the number of a production and j is the position of a special marker (here, a dot $\bullet$) to be inserted in the right part of P_i. Let $P_i = A \rightarrow \alpha$ for this discussion. An item $[i, j]$ is equivalent to the notation $[A \rightarrow \beta \bullet \gamma]$ where $\beta = \alpha_1\alpha_2 \cdots \alpha_j$ and $\gamma = \alpha_{j+1}\alpha_{j+2} \cdots \alpha_{n_i}$ (both notations are used in this thesis). If a state contains an item $[i, j]$ with $j < n_i$, then it has a *shift transition* for the symbol α_{j+1}, and α_{j+1} is called the shift symbol for that item. If a state contains an item $[i, n_i]$, then it has a *reduction* on P_i. (■ Tai called these *read transitions* and *reduce transitions* respectively.)

For any set I of items, let CLOSURE(I) be defined as the smallest set satisfying the following properties: (1) every item in I is in CLOSURE(I), and (2) if $[A \rightarrow \beta \bullet B\gamma]$ is in CLOSURE(I) and $B \rightarrow \delta$ is a production, then the item $[B \rightarrow \bullet \delta]$ is in CLOSURE(I).

Notes on the Notation of the Parser Generation Algorithms

The construction algorithm for SLR(1) parsers, below, closely follows the notation used by Tai. There are more elegant notations for describing this algorithm, but they do not work well for the NSLR(1) parser generators that appear later. Using the same notation for all algorithms facilitates the task of comparing the algorithms.

The notation used is precise, but not easily read. A few reminders and comments will help the unfamiliar reader significantly. Remember that:

$p = |P| =$ the number of productions in the grammar,

$n_i =$ the length of production P_i, and so

$[i, n_i] =$ a complete item for production P_i.

For example, if $P_5 = A \rightarrow \alpha$ then $[5, n_5] = [A \rightarrow \alpha \bullet]$. Complete items generate **reduce** actions in the current state.

Symbol sets attached to complete items are indexed by the production number. For example L_i is the lookahead set for the complete item $[i, n_i]$ constructed from production P_i. This indexing scheme can be used since there can be only one complete item generated by any given production in any particular state. It is not convenient to use the item number as an index, since the item set that forms the current state grows during parser construction. Indexing by production number is used for the symbol sets L_i, LM_i, F_i, and FR_i.

The set L is distinct from the sets L_i for $1 \leq i \leq p$. The set L is the set of symbols on which shift transitions are indicated for the current state.

For the state s and the lookahead symbol Y, the generated function $f(s, y)$ give the parser action to be performed, and the generated function $g(s, y)$ gives the destination state if the action is **shift**.

Algorithm PG_0: The Construction Algorithm for SLR(1) Parsers

(1) Initially, let $s_0 =$ CLOSURE({[0, 0]}) and $Q = \{s_0\}$ with s_0 "unmarked."

(2) For each unmarked state s in Q, mark it by performing the following steps:

(2.1) Compute the SLR(1) lookahead sets:

(2.1.1) For each i, $0 \leq i \leq p$, let

$$L_i = \begin{cases} \text{T\_FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

be the *simple 1-lookahead set* associated with the reduction on P_i.

(2.1.2) Let $L = \{Y \mid s$ contains $[i, j]$, $j < n_i$, $P_i = A \rightarrow \alpha$, and $\alpha_{j+1} = Y\}$ be the *simple 1-lookahead set* associated with all shift transitions from s.

(2.1.3) If L, L_0, L_1, "...," and L_p are not pairwise disjoint, then G is not SLR(1).

(2.2) For each terminal b in L, set $f(s, Y) =$ **shift**. If $L_0 = \{\dashv\}$, then set $f(s, \dashv) =$ **accept**. For each b in L_i, $1 \leq i \leq p$, set $f(s, Y) =$ **reduce** i. For each b in $V \cup \{\dashv\}$ but not in $L \cup L_0 \cup L_1 \cup \cdots \cup L_p$, set $f(s, Y) =$ **error**.

(2.3) For each b in L, compute GOTO(s, Y) as follows:

$$\text{GOTO}(s, Y)=\text{CLOSURE}(\{[i, j+1] \mid s \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow \alpha, \text{ and } \alpha_{j+1} = Y\}).$$

If GOTO(s, Y) is not already in Q, then add it to Q as an unmarked state. Set $g(s, Y) =$ GOTO(s, Y).

(2.4) For each Y in V but not in L, set $g(s, Y) =$ **error**. □

Any state with shift transitions only is called a *shift* state. Any state with exactly one reduction and no shift transition is called a *reduce* state. States which are neither shift states nor reduce states are called *inadequate* states. A CF grammar G is said to be *simple LR(1)* or *SLR(1)* if and only if every inadequate state in the SLR(1) parser for G has pairwise disjoint simple 1-lookahead sets associated with its shift transitions and reductions.

Algorithm P_0: The SLR(1) Parser

Initially, the stack contains the initial state s_0.

(1) Determine the current input symbol b.

(2) Let s be the state at the top of the stack.

(2.1) If $f(s, b) =$ **shift**, then remove b from the input, push the state $g(s, b)$ on to the stack, and go to (1).

(2.2) If $f(s, b) =$ **reduce** i, where $P_i = A \rightarrow \alpha$, then pop n_i states from the stack. A new state s' is then exposed as the top of the stack. Push the state $g(s', A)$ onto the stack and go to (1).

(2.3) If $f(s, b) =$ **error**, then halt and declare error.

(2.4) If $f(s, b) =$ **accept**, then halt and declare acceptance. □

4.4. The Original NSLR(1) Parser and Parser Generator

In order to permit the reading of this chapter without constant reference to Tai's paper, the original algorithm for generating NSLR(1) parsers is reproduced here, and is called Algorithm NPG_0. But first we present a short overview of NSLR(1) parsing.

The NSLR(1) parser generation algorithm NPG_0 generates parse tables for an NSLR(1) parser.* As illustrated in Figure 4.1, an NSLR(1) parser uses two stacks, a *state stack* and a *lookahead stack*. (■ Tai called these stacks the *state stack* and the *symbol stack*, respectively.) The principal differences from an ordinary SLR(1) parser are the addition of the lookahead stack, and the modification of the parser **reduce** action to make use of that stack. The top of the symbol lookahead contains the lookahead symbol for the parser. If the lookahead stack is empty the next symbol will be taken from the input stream. When implementing a character-level grammar, the input symbols are always single characters.

* An NSLR(1) parser can actually execute a wide variety of canonical and noncanonical LR(1) parse tables, but for simplicity we consistently use the term "NSLR(1) parser".

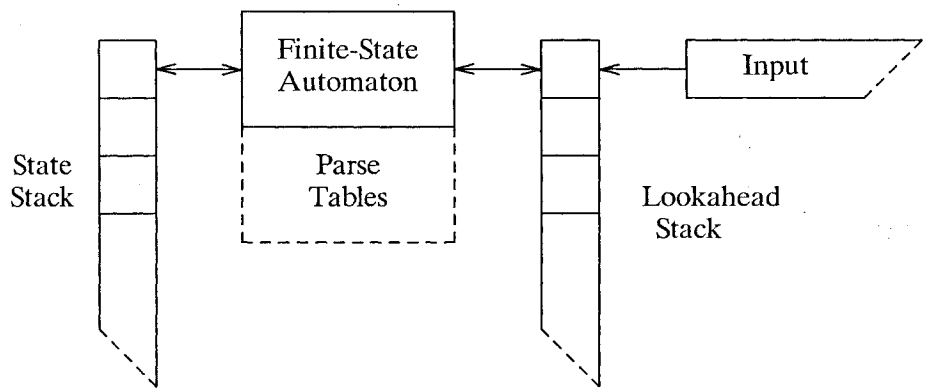

Figure 4.1: Schematic representation of an NSLR(1) parser. The principal differences from an SLR(1) parser are the addition of a lookahead stack, and the redefinition of the **reduce** action to make use of that stack.

There is slightly different way, used by Tai, to describe the NSLR(1) parser. In this second description, the parser is initiated with all the input stacked on the lookahead stack, with the front of the input at the top of the stack. This description is functionally identical to the one given above, but it simplifies the specification of the parsing algorithm. In the second description, popping the lookahead stack encompasses the operations of popping the lookahead stack and reading the next input character if the lookahead stack is empty. As a result, we use the second description in the rest of this thesis.

An NSLR(1) parser has only the four usual parser actions, **shift**, **reduce** i, **error**, and **accept**, but the operation of the **reduce** action has been modified. Let production $P_i = A \rightarrow \alpha$. The original **reduce** i action popped n_i states from the state stack exposing state s'. Then it pushed the state $g(s', A)$ onto the state stack. The new **reduce** i action pops n_i states from the state stack, and pushes the symbol A onto the lookahead stack.

An NSLR(1) parser can execute SLR(1) parse tables correctly with only slight modification. Since after the action **reduce** i, the symbol A on top of the lookahead stack will be the new lookahead symbol, and since s' will be the new current state, the next transition performed by the parser must be to the state $g(s', A)$, exactly the destination of the SLR(1) **reduce** i action. The only change needed to the parser generator is to ensure that the parser action $f(s',A)$ is a **shift** action. This simple change can be effected by changing step (2.2) of Algorithm PG_0 so that $f(s,Y)$ = **shift** for all shift symbols Y in a state rather than only for terminals.

NSLR(1) parser generation is an extension of SLR(1) parser generation. Initially a standard SLR(1) parser is generated, state by state, with the change given above. When a conflict arises in a state, a state-expansion algorithm is invoked. All conflicts involve a **reduce** action on a complete item, so the parser generator resolves conflicts, when it can, by eliminating the **reduce** action for conflicting lookahead symbols. The parser generator ensures that the state contains **shift** actions for those lookahead symbols that caused the conflict. The intention is to shift those lookahead symbols onto the state stack so that they can be reduced based on further lookahead. The reduced symbol will be pushed back onto the lookahead stack to serve as higher-level lookahead. For many grammars, this reduced symbol provides the extra lookahead information needed to resolve the original conflict.

A short example will clarify this discussion. Consider grammar $G_{4.1}$ shown in Figure 4.2. The language described by grammar $G_{4.1}$ can be given by the regular expression $c^*a \mid c^*b$. Although $L(G_{4.1})$ is regular, the grammar itself is not LR(k) for any k. Grammar $G_{4.1}$ is, however, SLR(1).

$$
\begin{aligned}
S &\rightarrow Aa \mid Bb \\
A &\rightarrow CA \mid C \\
B &\rightarrow DB \mid D \\
C &\rightarrow c \\
D &\rightarrow c
\end{aligned}
$$

Figure 4.2: Grammar $G_{4.1}$, a short SLR(1) grammar that is not LR(k) for any k.

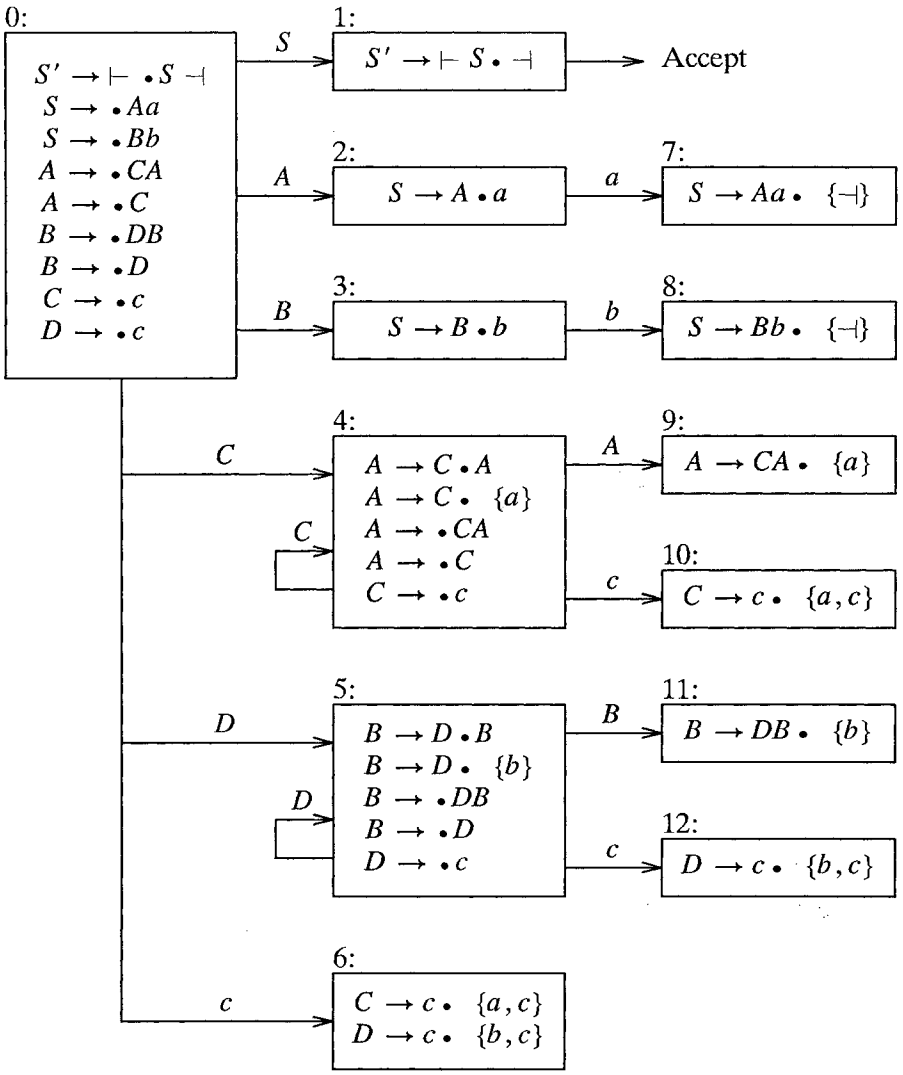

Figure 4.3: SLR(1) parsing automaton for grammar $G_{4.1}$.

The SLR(1) parsing automaton for this grammar, generated by Algorithm PG_0 is given in Figure 4.3. Notice that there is a conflict in state 6. When the parser is in state 6, and the lookahead symbol is c, two reductions are indicated: $[C \rightarrow c \bullet]$, and $[D \rightarrow c \bullet]$. The NSLR(1) parser generator examines the complete sets of symbols that can follow C and D in any sentential form and finds that they are $\{A, C, a, c\}$ and $\{B, D, a, c\}$, respectively. It finds that in these two sets there are no common nonterminals that can generate c. In other words c is in FIRST(C) and FIRST(D), but neither C nor D is common to both FOLLOW sets.

As a result of this analysis, the parser generator generates shift items that shift the symbol c with the intention of reducing it to C or D, and it deletes c from the lookahead sets for both reductions. In this way, the **reduce** actions on lookahead c are delayed, the lookahead c is reduced to either a C or a D, based on further lookahead, and the reduced symbol is used to resolve the conflict. The noncanonical expansion for state 6 is shown in Figure 4.4.

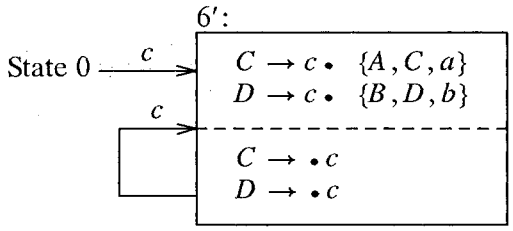

Figure 4.4: NSLR(1) expansion of state 6 of the SLR(1) automaton for grammar $G_{4.1}$.

A similar parsing problem can be described in terms of a character-level grammar for Pascal. After recognizing a type declaration in a Pascal program, if the lookahead character is "v", the parser cannot know whether the "v" is the first character of a the name of the next type variable to be declared, or is the first character of the keyword "var". As a result it cannot know whether to reduce all the type declarations seen so far to a type declaration section, or shift on the "v" to recognize a new type declaration. By shifting on the "v" and later context, it can recognize the input as either a keyword or a variable, reduce that later context, and have a reduced symbol as the lookahead symbol. Thus it delays the parsing decision until it has examined later context. (This scenario assumes that some method of disambiguating keyword from identifiers has been implemented. Such a method is presented in the next chapter.)

The method described above fails if the conflicting lookahead symbol is needed in its unreduced form for a **shift** action by another state. This would happen if grammar $G_{4.1}$ contained the rule $B \rightarrow Dc$ instead of $B \rightarrow D$. Then the symbol D could be followed by an unreduced string starting with c, and reducing a lookahead of c to C or D would block this later shift action. Algorithm NPG_0 uses the function LM_FOLLOW, defined below, to determine which symbols may be needed for shifting by predecessor states. Section 4.7.1 on the correctness of Algorithm NPG_1 contains more details on the functioning of Algorithm NPG_0.

The NSLR(1) parser generation algorithm and the NSLR(1) parsing algorithm, are reproduced here as they appeared in the original paper, but with some minor notational changes, and are called Algorithm NPG_0 and Algorithm NP_0, respectively. First some functions must be defined.

In addition to the function T_FOLLOW, defined previously, algorithm NPG_0 also uses the functions FOLLOW, LM_FOLLOW, and ϵ-CLOSURE. The function FOLLOW(A), for a nonterminal A, is the set of symbols that can follow A in some sentential form, and if A can be the rightmost symbol of a sentential form, then $\dashv$ is included in FOLLOW(A). That is,

$$\text{FOLLOW}(A) = \{Y \text{ in } V \cup \{\dashv\} \mid S' \stackrel{*}{\Rightarrow} \beta A Y \gamma \text{ for some } \beta \text{ in } \vdash V^* \text{ and } \gamma \text{ in } V^* \dashv\}.$$

(Thus the function T_FOLLOW(A) can also be defined as T_FOLLOW(A) = $\{a \text{ in } V_T \cup \{\dashv\} \mid a \text{ in FOLLOW}(A)\}$). LM_FOLLOW($A$), for a nonterminal A, is the set of symbols that can follow A in some left-sentential form, and if A can be the rightmost symbol of a left-sentential form, then $\dashv$ is included in LM_FOLLOW(A). That is,

$$\text{LM\_FOLLOW}(A) = \{Y \text{ in } V \cup \{\dashv\} \mid S' \stackrel{*}{\underset{lm}{\Rightarrow}} \beta A Y \gamma \text{ for some } \beta \text{ in } \vdash V^* \text{ and } \gamma \text{ in } V^* \dashv\}.$$

Note that LM_FOLLOW(A) $\subseteq$ FOLLOW(A). The function ϵ-CLOSURE is defined as:

For any set I of items, let ϵ–CLOSURE(I) be defined as the smallest set satisfying the following properties: (1) every item in I is in ϵ–CLOSURE(I), and (2) if $[A \rightarrow \alpha \bullet]$ is in I and FOLLOW(A) contains B such that $B \rightarrow \epsilon$ is in P, then $[B \rightarrow \bullet]$ is in ϵ–CLOSURE(I).

Algorithm NPG$_0$: Original Construction Algorithm for NSLR(1) Parsers

(1) Initially, let s_0 = CLOSURE({[0, 0]}) and $Q = \{s_0\}$ with s_0 "unmarked."

(2) For each unmarked state s in Q, mark it by performing the following steps:

(2.1) Compute the SLR(1) lookahead sets:

(2.1.1) For each i, $0 \leq i \leq p$, let

$$L_i = \begin{cases} \text{T\_FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

be the *simple 1-lookahead set* associated with the reduction on P_i.

(2.1.2) Let $L = \{Y \mid s \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow \alpha, \text{ and } \alpha_{j+1} = Y\}$ be the *simple 1-lookahead set* associated with all shift transitions from s.

(2.1.3) If L, L_0, L_1, "...," and L_p are pairwise disjoint, then s is SLR(1) consistent, so go to step (2.7). [Steps (2.2) to (2.6) comprise the *state expansion* part of the algorithm performed only on inconsistent states.]

(2.2) Let $s = \epsilon$–CLOSURE(s), so as to add items for new reductions. (This step is unnecessary if G is ϵ-free.)

(2.3) For each i, $0 \leq i \leq p$,

$$LM_i = \begin{cases} \text{LM\_FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

$$F_i = \begin{cases} \text{FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

(2.4)′ Let $R = \{Y \mid s \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow \alpha \text{ and } \alpha_{j+1} = Y\}$ be the set of shift symbols of s. For each i, $0 \leq i \leq p$,

(2.4.1)′ Compute the largest subset R_i of F_i that can distinguish the reduction by P_i: $R_i = \{Y \text{ in } F_i \mid Y \text{ is neither in } R \text{ nor in } F_j, \text{ where } 0 \leq j \leq p \text{ and } j \neq i\}$.

(2.4.2)′ Let $L_i = LM_i \cup R_i$ be the *NSLR(1)-lookahead set* associated with the reduction on P_i.

(2.4.3)′ Add the set I_i of new items to s, where $I_i = \{[q, 0] \mid P_q = B \rightarrow \beta, B \text{ is in } F_i, \beta \neq \epsilon \text{ and } \beta_1 \text{ is not in } L_i\}$.

(2.5)′ Let $L = \{Y \mid s \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow \alpha, \text{ and } \alpha_{j+1} = Y\}$ be the *NSLR(1)-lookahead set* associated with all shift transitions from s.

(2.6)′ If L, L_0, L_1, "...," and L_p are not pairwise disjoint, then abort (thus G is not noncanonical SLR(1)).

(2.7) For each Y in L, set $f(s, Y) =$ **shift**. If $L_0 = \{\dashv\}$, then set $f(s, \dashv) =$ **accept**. For each Y in L_i, $1 \leq i \leq p$, set $f(s, Y) =$ **reduce** i. For each Y in $V \cup \{\dashv\}$ but not in $L \cup L_0 \cup L_1 \cup \cdots \cup L_p$, set $f(s, Y) =$ **error**.

(2.8) For each Y in L, compute GOTO(s, Y) as follows:

$$\text{GOTO}(s, Y) = \text{CLOSURE}(\{[i, j+1] \mid s \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow \alpha, \text{ and } \alpha_{j+1} = Y\}).$$

If GOTO(s, Y) is not already in Q, then add it to Q as an unmarked state. Set $g(s, Y) = \text{GOTO}(s, Y)$.

(2.9) For each Y in V but not in L, set $g(s, Y) =$ **error**. □

An explanation of the notation used by Algorithm NPG_0 is in order. The subscript i of the sets FR_i, F_i, R_i, and L_i is a rule number. This technique is possible because these sets are associated only with complete items in a state and any rule can produce only one unique complete item. Ideally the subscripts would refer to an item number, but there is no convenient way to number items in a state. The alternative of attaching four separate sets to each item is equally unattractive.

The set R contains all shift symbols in the current set s. In Tai's paper, shift transitions were called read transitions and R is the first letter of *read*. Unfortunately R is also the first letter of *reduce*, and the reader should note that the sets R_i have no relation to the set R. Using S (for *shift*) for this set does not solve the problem as this letter is also heavily used, most notably as the start symbol. Here we simply keep Tai's naming convention and add this explanation in the hope that confusion is minimized.

Algorithm NP_0: The NSLR(1) Parser

An NSLR(1) parser needs two pushdown stacks SYMBOL_STK and STATE_STK. Initially, SYMBOL_STK contains the input string and STATE_STK contains the initial state s_0.

(1) Determine the symbol Y at the top of SYMBOL_STK.

(2) Let s be the state at the top of STATE_STK.

(2.1) If $f(s, Y) =$ **shift**, then pop Y from SYMBOL_STK, push the state $g(s, Y)$ on to STATE_STK, and go to (1).

(2.2) If $f(s, Y) =$ **reduce** i, where $P_i = A \rightarrow \alpha$, then pop n_i states from STATE_STK, push A onto SYMBOL_STK, and go to (1).

(2.3) If $f(s, Y) =$ **error**, then halt and declare error.

(2.4) If $f(s, Y) =$ **accept**, then halt and declare acceptance. □

4.5. More Parsing Terminology

Now that the SLR(1) and NSLR(1) parser generation and parser Algorithm have been presented, more terminology can be defined. As before, differences from standard notation are marked by a filled triangle (▸), and differences from Tai's notation are marked by a filled box (■).

A *core set* of items is a set of items without the lookahead sets specified. In an SLR parser and an NSLR parser all the states have unique core sets. ▸ A *semicomplete item* is an item whose dot appears neither at the beginning nor the end of the right part, i.e. items of the form $[A \rightarrow \alpha \bullet \beta]$ in which $\alpha \neq \epsilon$ and $\beta \neq \epsilon$. A *kernel item* is an item $[A \rightarrow \alpha \bullet \beta]$ in which $\alpha \neq \epsilon$. Thus both semicomplete and complete items are kernel items. The kernel set of a state is characteristic of that state in that all the items of a state can be reconstructed from the kernel set; applying CLOSURE and, if necessary, state expansion to the kernel set will generate all other items in the state. ▸ The term *state expansion* refers to the process of adding items to an SLR(1) inconsistent state to initiate reduction of right context that causes a conflict. In Algorithm NPG_0 state expansion is performed by steps 2.2 to 2.6.

▸ A *closure item* is an item generated by the CLOSURE function from a kernel item, and has its dot at the beginning of the right part of the rule. ▸ An *expansion item* is an item generated by state expansion. In an SLR parser, all items are either kernel items or closure items. ▸ An item with the dot at the beginning of the right part is called an *initial item*, and may be either a closure item or

an expansion item. An ϵ-reducing item, also called an ϵ-item for short, is an item formed from an ϵ-production. Such an item is represented by the notation $[A \rightarrow \bullet\,]$.

4.6. An Error in Algorithm NPG_0 in the Computation of NSLR Lookahead Sets

An error exists in Algorithm NPG_0 in the computation of lookahead sets for complete items. The problem is that lookahead sets for complete 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.

An example of this, devised by Thomas Pennello,[47] is presented here. Consider grammar G_{err} shown in Figure 4.5. Figure 4.6 shows the SLR(1) parsing automaton for this grammar. Notice that state 6 is inconsistent due to a reduce-reduce conflict on lookahead symbol g. NSLR state expansion modifies state 6 and produces the expanded state 6′ and the new state 13 as shown in Figure 4.7.

$$
\begin{aligned}
S &\rightarrow AF \mid BG \\
A &\rightarrow D \\
B &\rightarrow E \\
D &\rightarrow c \\
E &\rightarrow c \\
F &\rightarrow ga \\
G &\rightarrow gb
\end{aligned}
$$

Figure 4.5: Grammar G_{err} that illustrates an error in original NSLR(1) algorithm, NPG_0.

Figure 4.8 presents a parse of the legal sentence cga. In this figure, symbols on the state stack are subscripted by the number of the state in which the parser will be when that symbol is on top of the state stack. The terminal c is pushed on the state stack and the potential reduction of c to D is postponed until later context is shifted and reduced. The terminals g and a are shifted and reduced to F. Next c is reduced to D with look-ahead F, and D is shifted onto the state stack. At this point D should be reduced to A with lookahead F, but the reduce action contains only g as the lookahead symbol (state 4 in Figure 4.6). Thus the parse terminates with an error.

The error can be generalized as follows. Noncanonical state expansion delays the reduction of some symbols until later context is examined. The later context is reduced to some nonterminal A and pushed back onto the lookahead stack to serve as lookahead for a reduction in an expanded state. The lookahead sets for complete items in the expanded state, include nonterminal lookahead symbols. Some symbols on the state stack with lookahead A will be reduced to B, and B will be pushed onto the lookahead stack. The reduction to B will return the parser to a state where B may be shifted onto the state stack exposing A on the lookahead stack. The **shift** action may bring the parser into an unexpanded state r with complete items in which B is the last symbol of one of the complete items. Since r is unexpanded, however, the parser generation algorithm will have produced lookahead sets for these complete items that contain only terminals. As shown by G_{err}, a reduction may be the next correct action to take, but the lookahead stack now contains a nonterminal, and no reduction is possible. Hence the parse will fail.

4.7. Improved Handling of ϵ-Productions

Before the error described above can be fully corrected, the handling of ϵ-productions during state expansion must be improved. The improved handling of ϵ-productions is needed for the proof of correctness of Algorithm NPG_3. In any case, with this improvement more grammars are accepted. The change is especially important for character-level grammars for scannerless parsers of programming languages. In such grammars optional white space is ubiquitous, and the proper handling of this ϵ-production in particular ($optional\_white \rightarrow \epsilon$) is vital.

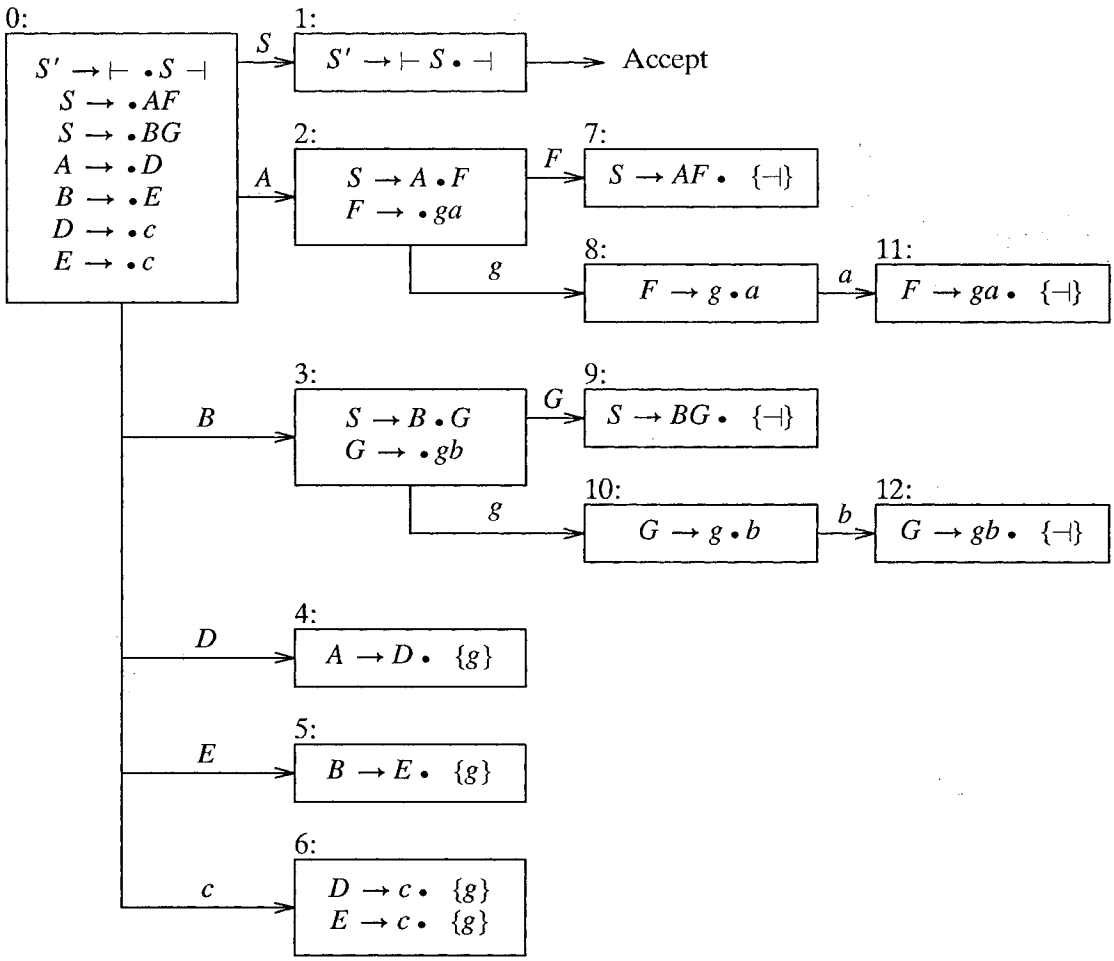

Figure 4.6: SLR(1) parsing automaton for G_{err}.

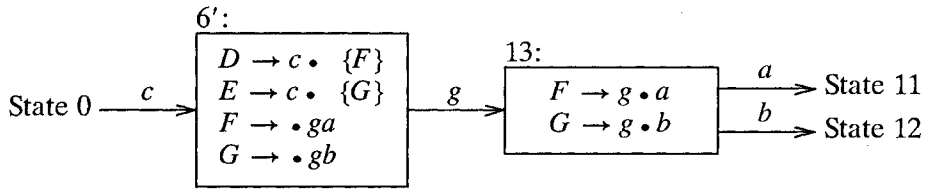

Figure 4.7: NSLR(1) expansion of SLR(1) automaton for G_{err}.

In Algorithm NPG_0, an inconsistent state is expanded by first forming the ϵ-CLOSURE of the state, and then generating new shift items for all conflicting symbols in the lookahead sets of complete items. Forming the ϵ-CLOSURE of a state consists of adding a new complete item $[B \rightarrow \bullet]$ for every symbol B in the lookahead set of any complete item, if B can produce ϵ. Forming the ϵ-CLOSURE(s) guarantees that all complete items that could possibly be needed for the expanded state will be included before conflicting lookahead symbols are identified. (All new items have the dot at the beginning of their right part, and all complete items have the dot at the end, so only ϵ-reducing items can have the dot both at the beginning and at the end of their right part.) As a result, the analysis of lookahead conflicts and the addition of new shift items for state expansion can be done once per state since the process can add no further complete items. Only new complete items could introduce new conflicts.

State Stack	Lookahead Stack
$\vdash$	$cga \dashv$
$\vdash c_{6'}$	$ga \dashv$
$\vdash c_{6'}\ g_{13}$	$a \dashv$
$\vdash c_{6'}\ g_{13}\ a_{11}$	$\dashv$
$\vdash c_{6'}$	$F \dashv$
$\vdash$	$DF \dashv$
$\vdash D_4$	$F \dashv$
Error!	

Figure 4.8: Sample parse of valid sentence *cga*. Current parser state is shown as a subscript of symbol on state stack. The error occurs because state 4 requires a lookahead of g to reduce D to A.

It is interesting to note at this point that when state expansion introduces an ϵ-item into a state, the only reason that the item can cause new conflicts is because a simple lookahead scheme is being used, rather than a full LR or LALR lookahead scheme. Suppose for instance that the ϵ-item $[D \rightarrow \bullet\,]$ was introduced into a state due to state expansion from the items $[A \rightarrow \alpha \bullet\,]$ $\{B, C, D, c\}$ and $[C \rightarrow \bullet Dc]$. Since the symbol D can produce ϵ, and D is in the lookahead set of A, all of the symbols that can actually follow D in this context are already in the lookahead set for A. But since a simple lookahead scheme is being used, FOLLOW(D), the lookahead set of the item $[D \rightarrow \bullet\,]$ may contain some symbols not in FOLLOW(A), and these symbols may cause new conflicts.

The disadvantage of the approach of Algorithm NPG_0, however, is that ϵ-CLOSURE adds some new items that are not needed to reduce further right context. As a result, needless new conflicts may be introduced and these conflicts may be unresolvable, resulting in the unnecessary rejection of the input grammar.

We propose Algorithm NPG_1 to improve the handling of ϵ-productions. In Algorithm NPG_1, the only ϵ-items added reduce ϵ to a shift symbol of an item that is needed to shift on conflicting right context. As a result, Algorithm NPG_1 has the two desirable properties that all symbols reduced from ϵ are immediately shifted by the same state (the property needed by the proof of Lemma 4.8.3), and all ϵ-reducing items contribute to the reduction of conflicting right context.

Two new functions are needed by Algorithm NPG_1: the functions FRONT and FR_FOLLOW. The function FRONT which is defined as

$$\text{FRONT}(\alpha) = \{X \mid \alpha_i{=}X \text{ and if } i \neq 1 \text{ then } \alpha_j \stackrel{*}{\Rightarrow} \epsilon \text{ for } 1 \leq j < i\}.$$

In other words, FRONT(α) contains the set of leading symbols in the string α up to and including the first one that does not generate ϵ. The function FRONT differs from the function FIRST in that FRONT(α) contains only symbols in α, whereas FIRST(α) may also contain symbols in sentential forms generated by α.

The function FR_FOLLOW stands for *fully-reduced FOLLOW*, and is defined as:

$$\text{FR\_FOLLOW}(A) = \{Y \text{ in } V \cup \{\dashv\} \mid S' \underset{lm}{\stackrel{*}{\Rightarrow}} \beta A \delta Y \gamma \text{ for some } \beta \text{ in } \vdash V^*, \gamma \text{ in } V^* \dashv, \text{ and } \delta \text{ in } V_N^* \text{ such that } \delta \stackrel{*}{\Rightarrow} \epsilon\}.$$

The similarity of FR_FOLLOW to Tai's function LM_FOLLOW is very strong, the only difference being that the symbols A and Y, of the definition, may have intervening ϵ producing symbols δ.

An alternate definition of FR_FOLLOW gives a better indication of how this function should be computed and helps to clarify its purpose.

$$\text{FR\_FOLLOW}(A) = \{Y \text{ in } V \cup \{\dashv\} \mid \text{there is a rule } C \rightarrow \beta B \delta Y \gamma \text{ where } A \text{ is in LAST}(B) \text{ and } \delta \stackrel{*}{\Rightarrow} \epsilon.$$

This second definition tells us that if there were a state s with a conflict on the lookahead symbol Y, and A was the symbol preceding the dot in some item in s, then we should not try to reduce Y to some nonterminal in order to resolve the conflict, as Y may be needed in its unreduced form for a shift in some other state. In other words, the symbol Y appears in a rule after a symbol that generates A, and therefore there will be a shift action on Y somewhere in the parser. That shift action may be involved in the current derivation, and if Y is reduced to some other symbol as lookahead, it will not be available for shifting.

Although FR_FOLLOW(A) is a superset of LM_FOLLOW(A), the computation of $\bigcup_i L_i$ using LM_FOLLOW in algorithm NPG_0 will be a superset of the same union computed using FR_FOLLOW in Algorithm NPG_1. The reason for this fact is that Algorithm NPG_0 first inserts ϵ-items for every ϵ-producing symbol in the lookahead set of any complete item. This action simulates the computation of FR_FOLLOW, but may include unnecessary items that enlarge $\bigcup_i L_i$.

Algorithm NPG$_1$: Improved handling of ϵ-productions.

Replace steps (2.2), (2.3), and (2.4)′ of Algorithm NPG_0 with these steps:

(2.2)′′ For each i, $0 \leq i \leq p$,

$$FR_i = \begin{cases} \text{FR\_FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

$$F_i = \begin{cases} \text{FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

(2.3)′′ Let $R = \{Y \mid s \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow \alpha \text{ and } \alpha_{j+1} = Y\}$ be the set of shift symbols of s.

(2.4)′′ For each i, $0 \leq i \leq p$,

(2.4.1)′′ Compute the largest subset R_i of F_i that can distinguish the reduction by P_i: $R_i = \{Y \text{ in } F_i \mid Y \text{ is neither in } R \text{ nor in } F_j, \text{ where } 0 \leq j \leq p \text{ and } j \neq i\}$.

(2.4.2)′′ Let $L_i = FR_i \cup R_i$ be the *NSLR(1)-lookahead set* associated with the reduction on P_i.

(2.4.3)′′ Form the set I_i of new items where $I_i = \{[q, 0] \mid P_q = B \rightarrow \beta, B \text{ is in } F_i, \beta \neq \epsilon, \text{ and for some } X \text{ in FRONT}(\beta), X \text{ is in } F_i - L_i\}$.

(2.4.4)′′ For each unmarked item $[q, 0] = [B \rightarrow \bullet \beta]$ in I_i, such that $C = \beta_1$ is in V_N, and there is a rule $P_r = C \rightarrow \gamma$, where $\gamma \stackrel{*}{\Rightarrow} \epsilon$:

(2.4.4.1)′′ Add $[r, 0] = [C \rightarrow \bullet \gamma]$ to I_i.

(2.4.4.2)′′ Mark the item $[q, 0]$.

(2.4.5)′′ Add the set I_i to s. □

Steps (2.4.3)′′ and (2.4.4)′′ contain the principal differences of Algorithm NPG_1 from NPG_0. Step (2.4.3)′′ adds the new shift items needed to shift on conflicting right context. The right part of some of these items may begin with symbols that are not in conflict, but are reducible from ϵ. Step (2.4.4)′′ adds the items needed to reduce such symbols from ϵ.

When step (2.4)'' is done, the sets L_i contain the lookahead sets for each complete item $[i,n_i]$. All symbols in F_i but not in L_i will be shifted and reduced in order to resolve a conflict. After step (2.4)'' is finished, some conflicts introduced during state expansion may still be resolvable. With suitable but tricky manipulation of F_i and L_i it would be possible to repeat step (2.4)'' as many times as necessary, and resolve some of the new conflicts. The class of extra grammars accepted in this way, however, seems to be of very limited usefulness. As a result, the complexity added by repeating the conflict resolution was not considered warranted and was not included in Algorithm NPG_1.

4.7.1. Correctness of Algorithm NPG$_1$

We now prove some lemmas and a theorem about Algorithm NPG_1. The proofs are fairly informal, since the length of complete formal proofs would be excessive.

Lemma 4.7.1: All SLR(1) consistent grammars will be accepted by Algorithm NPG_1.

Proof: This proof is trivial. Unless a conflict arises in the grammar, Algorithm NPG_1 is identical to Algorithm PG_0, the SLR(1) parser generator. If such a conflict arose then Algorithm PG_0 would reject the grammar whereas Algorithm NPG_1 may accept it. Therefore the grammars accepted by Algorithm NPG_1 must be a superset of the grammars accepted by PG_0. □

Lemma 4.7.2: Only initial shift items, that is items with the dot at the beginning of the right part, need be added during state expansion; no semicomplete items need be generated.

Proof: The purpose of state expansion is to reduce ambiguous right context into a unique lookahead symbol. The reduced symbol will then be used as a lookahead symbol to select between one **reduce** action, and a conflicting **reduce** action or **shift** action.

Suppose a semicomplete item of the form $[A \rightarrow \alpha \bullet X\beta]$ were added to a state to shift and later reduce the lookahead symbol X. If there is no derivation $\alpha \stackrel{*}{\Rightarrow} \epsilon$, such an item would combine some left context with right context to produce a symbol A, therefore such an item need not be considered. If left context of α were valid in the current state, then the LR construction algorithm would have already included the item $[A \rightarrow \alpha \bullet X\beta]$ in the state. On the other hand, a semicomplete item of the same form, where there *is* a derivation $\alpha \stackrel{*}{\Rightarrow} \epsilon$, would not necessarily combine left context with right context and therefore is eligible for inclusion during state expansion. Notice, however, that adding the initial item $[A \rightarrow \bullet \alpha X\beta]$ along with all items needed to reduce ϵ to α would achieve the same purpose. This second choice is the one made by Algorithm NPG_1. The effects are equivalent, unless other initial items of the form $[B \rightarrow \gamma \bullet X\delta]$, were added where γ may be ϵ, but $\gamma \neq \alpha$. (If $\gamma = \alpha$ then both items could be added as initial items.) A grammar with two rules of the above form, however, will always cause an unresolvable conflict. Since $\alpha \stackrel{*}{\Rightarrow} \epsilon$, $\gamma \stackrel{*}{\Rightarrow} \epsilon$, and there are rules $A \rightarrow \alpha X\beta$, and $B \rightarrow \gamma X\delta$, the symbol X will be in FR_FOLLOW(C) for every symbol C in α and γ. Without loss of generality assume that $|\alpha| \leq |\gamma|$. Since the grammar contains no useless rules, there must be some state with the item $[\alpha_i \rightarrow \bullet]$ $\{X \ldots\}$, and and one of the items $[\gamma_i \rightarrow \bullet]$ $\{X \ldots\}$ or $[B \rightarrow \gamma \bullet X\delta]$. Either case leads to a conflict, and since X is in FR_FOLLOW(α_i) the conflict is unresolvable. □

Lemma 4.7.3: Algorithm NPG_1 adds all necessary items that shift lookahead symbols in conflict.

Proof: Using Lemma 4.7.2, we need only show that Algorithm NPG_1 adds all possible initial items of the form $[B \rightarrow \bullet \beta]$ that shift the lookahead symbol X for all X in conflict. Let the set $\underline{F} = \bigcup_i F_i$ for all i such that there is a complete item $[i, n_i]$ in the current state. Thus $\underline{F}$ contains all symbols, terminal and nonterminal, that could follow the left context on the state stack. (Since the *simple* strategy is used to compute the lookahead set, $\underline{F}$ may also contain some symbols that may not follow the left context.) Therefore the symbol B must be in $\underline{F}$. Let the set $\underline{FR} = \bigcup_i FR_i$ for all i such that there is an complete item $[i, n_i]$ in the current state. Thus $\underline{FR}$ contains all symbols that may not be shifted by an expanded state, since they may be needed for a **shift** action by a predecessor state.

(Since the *simple* strategy is used to compute $\underline{FR}$, it may also contain some symbols that will not be shifted by predecessor states.) Therefore if the symbol X is in $\underline{FR}$ it may not be correct to shift X and reduce it. An item may cause the shifting of lookahead symbol X by the current state, or by descendant states, if its right part β has the form $\delta X \eta$, where $\delta \stackrel{*}{\Rightarrow} \epsilon$. (If $\delta = \epsilon$ the X will be shifted by the current state, if not, then by some descendant state.) If X is in FRONT(β) then β has the desired form. Algorithm NPG_1 inserts, during state expansion, all items $B \rightarrow \beta$ such that B is in $\underline{F}$ and X is in β, unless X is in FR_i for some i. In this last case it may be incorrect to shift X so X is left in L_i to cause and unresolvable conflict, and rejection of the grammar.

If $\delta \neq \epsilon$, then the state must also contain items to reduce δ_1 from ϵ. (The symbols δ_j for $1 < j \leq |\delta|$ will be reduced from ϵ by descendant states.) Step (2.4.4.1)'' adds all possible items of the form $[\delta_1 \rightarrow \bullet \gamma]$ where $\gamma \stackrel{*}{\Rightarrow} \epsilon$. (In an unambiguous grammar, there will only be one such item. If there are two or more, a reduce-reduce conflict on X will result.) If $\gamma \neq \epsilon$ then the loop of step (2.4.4)'' will add the items needed to reduce γ_i from ϵ. □

Lemma 4.7.4: In states produced by Algorithm NPG_1, all symbols reduced from ϵ will be immediately shifted by the same state that reduced them. That is, if a state contains an item of the form $[A \rightarrow \bullet]$, it will also contain an item of the form $[B \rightarrow \alpha \bullet A \beta]$.

Proof: All ϵ-items are generated either by CLOSURE or by state expansion. If an ϵ-item $[A \rightarrow \bullet]$ was generated by CLOSURE then there must be a kernel item $[B \rightarrow \alpha \bullet A \beta]$ in the same state. This kernel item will shift the symbol A reduced from ϵ.

If the ϵ-item $[A \rightarrow \bullet]$ was generated by state expansion, then it was generated by step (2.4.4.1)''. In this case the item takes the form $[C \rightarrow \bullet]$, and was generated by step (2.4.4)'', and there must be an expansion item of the form $[B \rightarrow \bullet C \delta]$, which will shift C. □

Lemma 4.7.5: During state expansion, Algorithm NPG_1 may generate **shift** actions for lookahead symbols not in conflict. All such shift items will cause new unresolved conflicts, and hence it is not necessary to generate all possible shift items for those lookahead symbols.

Proof: Step (2.4.4.1)'' may generate items of the form $[C \rightarrow \bullet D \delta]$. In some cases these new state expansions items indicate a shift of the symbol D even though there was a unique reduction on lookahead symbol D indicted before state expansion. Since D is in FIRST(C), C is in FIRST(B), and B is in F_i, therefore D is also in F_i. This fact is true since if a symbol X is in F_i then by construction all the symbols in FIRST(X) are in F_i. If D is in F_i, and there was no conflict on lookahead D before state expansion, then D is also be in L_i, the ultimate look ahead set for a reduction by rule i. Since D is now in a shift lookahead set, and if D was not in conflict before state expansion it is still in a reduction lookahead set, therefore a conflict will result after state expansion. □

Lemma 4.7.6: If state expansion by Algorithm NPG_1 fails with an unresolvable conflict, then so will state expansion by Algorithm NPG_0.

Proof: States expanded by Algorithm NPG_1 differ from those expanded by Algorithm NPG_0 in only two ways: 1) some ϵ-items generated by NPG_0 are not generated by NPG_1, and 2) some shift items generated by NPG_1 are not generated by NPG_0.

Case 1: The state expansion by Algorithm NPG_0 adds all possible ϵ-items, hence states expanded by Algorithm NPG_1 can contain no ϵ-items that would not appear if the same state were expanded by Algorithm NPG_0. Nevertheless, since Algorithm NPG_0 resolves conflicts after adding ϵ-items, whereas Algorithm NPG_1 does not, it may seem that Algorithm NPG_0 may resolve some conflicts not resolved by NPG_1. This is not the case. Only step (2.4.4.1)'' of Algorithm NPG_1 adds ϵ-items. If an ϵ-item $[r, 0] = [C \rightarrow \bullet]$ is added, then there must be an item of the form $[B \rightarrow \bullet \beta]$ such that C is in FIRST(β_1), that was added by step (2.4.3)'' due to a conflict on some symbol X in FRONT(β), where $X \neq C$. Since all lookahead symbols in conflict before state expansion by NPG_1 would also be in conflict after ϵ-CLOSURE by NPG_0, therefore X

would be in conflict for Algorithm NPG_0 also. Since C is in FIRST(β) and X is in FRONT(β), X is in FR_r and hence in L_r, the ultimate lookahead set for the reduction $[r, 0]$. As a result there is an unresolvable conflict on lookahead symbol X in Algorithm NPG_0 too.

Case 2: State expansion by Algorithm NPG_1 adds some shift items not added by NPG_0. Such items take the form $[B \rightarrow \bullet \delta X \eta]$, where $\delta \stackrel{*}{\Rightarrow} \epsilon$ and the symbol X was in conflict before state expansion. All lookahead symbols in conflict for Algorithm NPG_1 before state expansion are also in conflict for NPG_0 after ϵ-CLOSURE. Furthermore, there is a symbol $C = \delta_1$ and a production $P_r = C \rightarrow \epsilon$ such that X is in FR_r. Therefore, if Algorithm NPG_1 generates a shift item not generated by Algorithm NPG_0, Algorithm NPG_0 will also always fail with an unresolvable conflict. □

Theorem 4.1: Algorithm NPG_1 accepts all grammars accepted by Algorithm NPG_0.

Proof: All differences between Algorithm NPG_0 and Algorithm NPG_1 are in the state expansion algorithm. By Lemma 4.7.6, if state expansion by Algorithm NPG_1 fails then so will state expansion by Algorithm NPG_0. Sometimes both state expansions may succeed, but produce different conflict-free expanded states. The expanded states may differ in two ways: 1) the state produced by Algorithm NPG_0 may have more ϵ-items than the one produced by Algorithm NPG_1, and 2) the state produced by Algorithm NPG_1 may have shift items not found in the state produced by Algorithm NPG_0. These differing states may result in differing states produced by GOTO.

Case 1: In a consistent state s, differences in ϵ-items do not affect the state produced by GOTO(s, Y), since the kernel set of destination states are formed solely from shift items in s. Therefore, such differences will not lead to differing conflicts in destination states.

Case 2: As shown in case 2 of the proof of Lemma 4.7.6, when Algorithm NPG_1 produces an expansion item not produced by Algorithm NPG_0, Algorithm NPG_0 fails and rejects the input grammar. As a result possible new conflicts in differing descendant states are not relevant.

Therefore all grammars accepted by Algorithm NPG_0 are also accepted by Algorithm NPG_1. □

4.7.2. New Grammars Accepted by Algorithm NPG_1

There are some grammars that would be rejected by Algorithm NPG_0 that are now acceptable. Consider for instance the grammar G_ϵ in Figure 4.9. This grammar describes the language $\{cca, ccja, ccb, ccjb, ic, ijc\}$

$$S \rightarrow AJa \mid BJb \mid iJD$$
$$A \rightarrow DD$$
$$B \rightarrow EE$$
$$D \rightarrow c$$
$$E \rightarrow c$$
$$J \rightarrow \epsilon \mid j$$

Figure 4.9: Grammar G_ϵ, a grammar that is rejected by the NSLR(1) parser construction Algorithm NPG_0, but is accepted by Algorithm NPG_1.

Figure 4.10(a) shows the state of the SLR(1) parser for G_ϵ that contains conflicts. There is a conflict between the two items $[D \rightarrow c \bullet]$ and $[E \rightarrow c \bullet]$ on lookahead c. Figure 4.10(b) shows the the same state after state expansion by Algorithm NPG_0. The ϵ-CLOSURE has brought in the new item $[J \rightarrow \bullet]$ $\{D, a\}$, since J can produce ϵ and J was in the lookahead set of the complete item $[D \rightarrow c \bullet]$. As a result, there is an unresolvable conflict on lookahead symbol D. This conflict cannot

be resolved by a **shift** action, because no other symbol in the lookahead set produces D. When Algorithm NPG_1 is used, the item $[J \rightarrow \bullet]$ will not be brought in, and a state without conflicts shown in Figure 4.10(c) will be produced by state expansion.

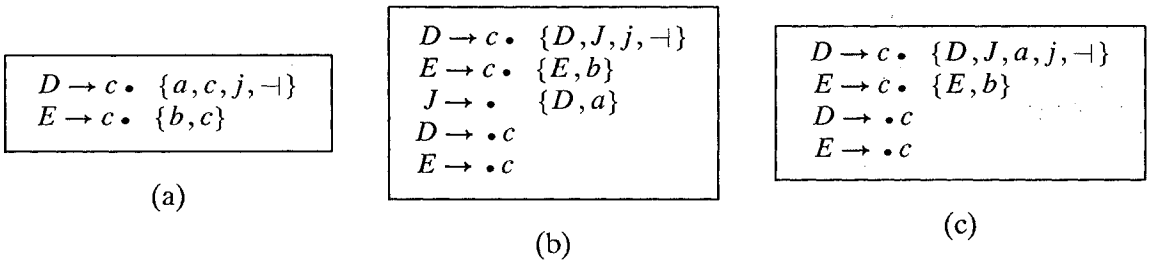

Figure 4.10: The state with conflicts from the parsing automaton for G_ϵ is shown (a) before state expansion, (b) after state expansion by Algorithm PG_0, and (c) after state expansion by Algorithm PG_1. A conflict on lookahead D remains in (b) but not in (c).

4.8. Algorithm NPG_2: Correcting the Lookahead Sets of Algorithm NPG_1

The error in the lookahead sets of consistent states pointed out in Section 4.6 has not been corrected in Algorithm NPG_1, so let us return to this problem. The simplest method of correcting the lookahead sets generated by Algorithm NPG_1 is to eliminate the function T_FOLLOW and use the function FOLLOW in its place. Let us call Algorithm NPG_1 corrected in this way, Algorithm NPG_2.

In Algorithm NPG_1, the only usage of T_FOLLOW is in computing the lookahead set for complete items in SLR(1) consistent states. The end result of the change will be to produce **reduce** actions in the parser action table for all possible following symbols, terminals and nonterminals, that appear in any sentential form of a derivation in the grammar. Once this is done all complete items in all states, expanded and unexpanded will have all possible follow symbols in their lookahead sets.

Although this correction is simple, the resulting parser generator may fail for grammars with *invisible* symbols.

Definition: The set of *invisible* symbols contains those nonterminals that can ultimately produce only ϵ.

$$V_I = \{A \text{ in } V_N \mid A \stackrel{*}{\Rightarrow} w \text{ implies } w = \epsilon\}.$$

Similarly the set of visible symbols contains those symbols that can produce terminals.

$$V_V = \{X \text{ in } V \mid X \stackrel{*}{\Rightarrow} x \text{ for } x \text{ in } V_T^+\}.$$

These definitions imply that:

$$V_V = V - V_I.$$

Algorithm NPG_2 fails for grammars containing invisible symbols in the sense that it even rejects some SLR(1) grammars that contain invisible symbols. Invisible symbols can be deleted from a grammar without changing the language produced; nevertheless, they are sometimes useful for writing grammars for programming languages when semantic actions are attached to the ϵ-productions that use them. In addition, Algorithm NPG_2 generates many entries in the parser action table that cannot be used in any parse. These two deficiencies of Algorithm NPG_2 are addressed in Sections 4.9 and 4.11 respectively.

4.8.1. Correctness of Algorithm NPG_2

Assume that the grammars to be processed by Algorithm NPG_2 contain no invisible symbols. The parser action table produced by Algorithm NPG_2 will contain all the entries of the table produced by Algorithm NPG_1 plus additional **reduce** actions on nonterminal lookahead symbols. This section proves that the added lookahead entries correct the lookahead error of Algorithm NPG_1, and do not introduce any new conflicts. This is done by proving two properties of the new entries:

1) the new entries do not make any formerly consistent states inconsistent,
2) the new entries, with the old entries, include all possible valid lookahead symbols.

The modification will, however, probably generate many useless entries.

Lemma 4.8.1: If a grammar contains no invisible symbols then using FOLLOW in place of T_FOLLOW in computing the lookahead sets for **reduce** actions will not make any formerly consistent state into an inconsistent state.

Proof: We now show that a state that was consistent according to Algorithm NPG_1 will remain consistent according to Algorithm NPG_2. This is all that is needed since once a state is found to be inconsistent it is processed by a state expansion algorithm that does not use T_FOLLOW and hence does not change from Algorithm NPG_1 to Algorithm NPG_2.

In an SLR(1) consistent state r, generated by Algorithm NPG_1, a complete item $[i, n_i]$ of the form $[A \rightarrow \alpha \bullet]$ will have lookahead set T_L_i = T_FOLLOW(A) associated with it. By Algorithm NPG_2, each complete item $[i, n_i]$ will have L_i = FOLLOW(A) as its lookahead set. Assume that the grammar has no invisible symbols, so that $V_I = \varnothing$. Then for every nonterminal B in L_i, FIRST(B) $\neq \varnothing$ and FIRST(B) $\supset T\_L_i$. Furthermore all terminals in L_i are also in T_L_i. A new conflict would be either a reduce-reduce conflict or a shift-reduce conflict.

Case 1: New reduce-reduce conflicts. A new reduce-reduce conflict would arise if two distinct complete items in some state r have disjoint terminal lookahead sets, but intersecting nonterminal lookahead sets. Formally, for some i and j such that $i \neq j$, and $[i, n_i]$ and $[j, n_j]$ are in r, then $T\_L_i \cap T\_L_j = \varnothing$, but $L_i \cap L_j \neq \varnothing$. Suppose there is a nonterminal C such that C is in $L_i \cap L_j$. Then by the definition of L_i, FIRST(C) $\subseteq L_i$ and FIRST(C) $\subseteq L_j$. But T_FIRST(C) $\subset$ FIRST(C), hence T_FIRST(C) $\subseteq L_i$ and T_FIRST(C) $\subseteq L_j$. But all terminals in L_i are also in T_L_i, hence T_FIRST(C) $\subseteq T\_L_i$ and T_FIRST(C) $\subseteq T\_L_j$, therefore

$$\text{T\_FIRST}(C) \subseteq (T\_L_i \cap T\_L_i).$$

In SLR(1) consistent states, however, $T\_L_i \cap T\_L_j = \varnothing$, hence T_FIRST($C$) $= \varnothing$, but since there are no invisible symbols in the grammar, this is impossible. Hence there is no C in $L_i \cap L_j$, and there are no new inconsistencies due to reduce-reduce conflicts.

Case 2: New shift-reduce conflicts. As in the original paper, let L be the simple 1-lookahead set for shift transitions from state r:

$$L = \{X \mid r \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow x, \text{ and } x_{j+1} = X\}$$

This definition can be rephrased as:

$$L = \{X \mid r \text{ contains } [A \rightarrow \alpha \bullet X\beta]\}$$

A new shift-reduce conflict will arise if $T\_L_i \cap L = \varnothing$, but $L_i \cap L \neq \varnothing$ for some i. This implies that there is some nonterminal A, such that A is in L_i and A is in L. Since A is in L and the SLR(1) consistency of state r is tested on the CLOSURE(r), we know that L contains A and $\{B \mid [C \rightarrow \bullet B\beta]$ is in r for all C in FIRST(A)$\}$. Since G has no invisible symbols, this recursive definition of L must include a terminal symbol a in T_FIRST(A). That symbol a must also be a member of T_L_i and a

conflict would have already existed in the unexpanded state. Hence no newly inconsistent state would be generated. □

Lemma 4.8.2: Algorithm NPG_2 will generate all lookahead entries for complete items that could possibly be needed by an NSLR two-stack parser.

Proof: Let us represent the state of an NSLR two-stack parser by the representation (α, β), where α is the contents of the state stack with the stack top at the right, and β is the contents of the lookahead stack with the stack top at left. We can represent an arbitrary **reduce** action that reduces the string δ to D by the transition $(\vdash\gamma\delta, X\theta) \rightarrow (\vdash\gamma, DX\theta)$, Where the string $X\theta$ ends with $\dashv$. Notice that the **reduce** action requires a lookahead symbol X in $(V \cup \dashv)$. This **reduce** action will correspond to the step $D \rightarrow \delta$ in the derivation

$$S' \stackrel{*}{\Rightarrow} \gamma DX\theta \Rightarrow \gamma\delta X\theta \stackrel{*}{\Rightarrow} \vdash x \dashv$$

By the definition of FOLLOW, Any X that can occur in such a derivation must be a member of FOLLOW(D). Hence using a lookahead set of FOLLOW(D) for the reduction of δ to D will include all possible lookahead symbols needed. □

It would also be desirable to prove that all lookahead entries generated by Algorithm NPG_2 will be needed for some NSLR(1) parse. Unfortunately this is not the case, since both Algorithms NPG_1 and NPG_2 generate many useless entries. Reducing the number of useless entries is the topic of Section 4.11.

Take note also of the word *simple* in the acronym NSLR. As in an SLR parser it implies that the lookahead sets may contain many lookahead symbols that cannot occur in valid strings of the language. These extra entries can cause some spurious **reduce** actions on invalid input strings, but never a spurious **shift** action. This characteristic of SLR parse table is discussed in more detail by Aho and Ullman[1] in Section 7.3.5. An NSLR parser will not only allow some invalid reductions on invalid inputs, but may also perform some **shift** actions on incorrect symbols. An NSLR parser will accept no invalid input strings, but an exact statement of how it behaves on erroneous strings is not as easily presented as for SLR parsers, and is not presented here. A study of behaviour of noncanonical parsers on erroneous input would make a good topic for future research.

4.9. Allowing Grammars with Invisible Symbols

The proof of Lemma 4.8.1 required that the grammar being processed contain no invisible symbols. Since invisible symbols can ultimately produce only ϵ they can be deleted from a grammar without changing the language produced. Occasionally, however, ϵ-productions using invisible symbols are useful in a parser when semantic actions are attached.

Algorithm NPG_2 can still produce correct tables for grammars containing invisible symbols, but in cases where the invisible symbols appear in the lookahead set of two complete items in a state, a conflict may needlessly be announced. The problem is illustrated by grammar G_{inv} defined as follows:

$$S \rightarrow ANa \mid BNb$$
$$A \rightarrow c$$
$$B \rightarrow c$$
$$N \rightarrow \epsilon$$

Grammar G_{inv} is an SLR(1) grammar, but once nonterminals are included in the lookahead sets of complete items, as per Algorithm NPG_2, a new conflict arises. In parsing the valid sentence ca, the parser cannot decide whether to reduce the initial c to A or B without inspecting the lookahead symbol. An SLR(1) parser generator will generate the complete items $[A \rightarrow c \bullet]$ with lookahead $\{a\}$ and $[B \rightarrow c \bullet]$ with lookahead $\{b\}$ for making this reduction. Since the lookahead sets contain only symbols from T_FOLLOW(A) and T_FOLLOW(B), there is no conflict. In Algorithm NPG_2, however, the lookahead sets will be $\{N, a\}$ and $\{N, b\}$ respectively, and a conflict on N results. The

conflict cannot be resolved by state expansion, and hence the grammar is rejected as not being NSLR(1). Algorithm NPG_3, a modification of Algorithm NPG_2, provides a way of processing grammars with invisible symbols without needlessly announcing conflicts.

4.9.1. Algorithm NPG$_3$: Correct Handling of Invisible Symbols

The method proposed here for handling lookahead conflicts on invisible symbols is simply to ignore them. The justification for this treatment, is that since invisible symbols never generate any actual terminal symbols in sentences of a grammar, they can be ignored. The proof of this is presented in the next subsection.

To correct Algorithm NPG_2 so that it accepts all SLR(1) grammars as NSLR(1) grammars, exclude invisible symbols from the sets L_i, FR_i, and F_i. In other words, these three sets should be initialized as:

$$L_i = F_i = \begin{cases} \text{FOLLOW}(A) \cap V_V & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

$$FR_i = \begin{cases} \text{FR\_FOLLOW}(A) \cap V_V & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

After applying these changes to Algorithm NPG_2, call it Algorithm NPG_3.

4.9.2. Proof of Correctness of Algorithm NPG$_3$

Algorithm NPG_3 has the same behavior as NPG_2 except that it ignores conflicts on invisible symbols. In this section we present and prove six lemmas and two theorems to show that conflicts on invisible symbols can be ignored in Algorithm NPG_2.

Theorem 4.2: In an NSLR(1) parser generated by Algorithm NPG_2, every **reduce** action that returns the parser to an unexpanded conflict-free state and pushes a nonterminal A onto the lookahead stack will be immediately followed by a shift transition that removes A from the lookahead stack, or by an accept transition that terminates the parse.

Proof: Every reduction of a string α to a symbol A, returns the parsing automaton to a state r with an initial item $[A \rightarrow \bullet \alpha]$. This is the item that initiated the shifting of the string α. In an unexpanded state, with only one possible exception, such an item must have been generated by CLOSURE from a kernel item of the form $[B \rightarrow \beta \bullet A\gamma]$. Such a kernel item indicates a shift transition out of r on lookahead A. Since the state is conflict-free, this must be the only action available on a lookahead of A.

The one initial item not generated by CLOSURE from a kernel item is $[S' \rightarrow \bullet \vdash S \dashv]$. A reduction to S', however, would indicate an accept action, terminating the parse. □

Corollary: A nonterminal A is unnecessary in the lookahead sets of complete items unless some expanded state in the parser contains an expansion item with a left part of A, and it contains no shift item of the form $[B \rightarrow \beta \bullet A\gamma]$.

This corollary will be used in section 4.11 to reduce the size of the lookahead sets.

Lemma 4.9.1: If a grammar with an invisible symbol A, has more than one distinct production with a left part of A, that grammar will be ambiguous, and will be rejected by Algorithm NPG_2 with unresolvable conflicts.

Proof: The ambiguity of the grammar is obvious. Suppose the grammar has the two rules $A \rightarrow \alpha$ and $A \rightarrow \beta$, where $\alpha \neq \beta$, and A is in V_I. Since the grammars under consideration have no useless symbols, there must be a derivation $S \stackrel{*}{\Rightarrow} \gamma A \theta$. This sentential form can generate two sentential forms $\gamma\alpha\theta$ and $\gamma\beta\theta$. Since α and β are both in V_I^+, both of these sentential forms must generate the same sentences.

To see that such a grammar will be rejected by Algorithm NPG_2, consider a state with an item $[B \rightarrow \delta \bullet A\theta]$. Since A is not useless there must be such a state. The CLOSURE operation on this state will generate the items $[A \rightarrow \bullet\alpha]$ and $[A \rightarrow \bullet\beta]$. If either α or β are not ϵ, further items will be generated. In the end there will be at least two items $[C \rightarrow \bullet]$, and $[D \rightarrow \bullet]$, where C and D are in FIRST(A). If $\alpha = \epsilon$ then $C = A$, otherwise C is in FIRST(α). Similarly, if $\beta = \epsilon$ then $D = A$, otherwise D is in FIRST(β). Since α and β are in V_I^+, C and D must also be in LAST(A). Since C and D are in LAST(A), every symbol in FR_FOLLOW(A) must also be in FR_FOLLOW(C) and FR_FOLLOW(D), and these symbols will lead to unresolvable conflicts on the complete items $[C \rightarrow \bullet]$ and $[D \rightarrow \bullet]$. □

Corollary: Every invisible symbol in a grammar accepted by Algorithm NPG_2 must either generate ϵ directly, $A \Rightarrow \epsilon$, or indirectly, $A \Rightarrow \alpha \stackrel{+}{\Rightarrow} \epsilon$, but may not do both.

Lemma 4.9.2: Shift-reduce conflicts on invisible symbols that directly produce ϵ can be resolved in favour of shifting. That is, if a state contains two items, a shift item of the form $[B \rightarrow \alpha \bullet A\beta]$, and a complete item of the form $[C \rightarrow \gamma \bullet]$ with a lookahead set that includes A; where A is in V_I, and $A \Rightarrow \epsilon$, then A can be deleted from the lookahead set of the complete item.

Proof: In a parsing automaton created by Algorithm NPG_2, if a state r contains the shift item $I = [B \rightarrow \alpha \bullet A\beta]$, where A is an invisible symbol such that $A \Rightarrow \epsilon$, then r must also contain a complete item $J = [A \rightarrow \bullet]$. Item J will have been generated from item I either by CLOSURE or by step (2.4.4)'' of state expansion. By Lemma 4.7.4, if a symbol A reduced from ϵ appears on the top of the lookahead stack for a state r then it was reduced by the same state r due to item J.

This lemma assumes a conflict on lookahead A between item I and a complete item $K = [C \rightarrow \gamma \bullet]$. Before the symbol A is reduced from ϵ by state r, some symbol X, part of further right context must appear on the stack, and a reduction to A chosen over a reduction to C based on that lookahead. If there was no conflict between the complete items $K = [C \rightarrow \gamma \bullet]$ and $J = [A \rightarrow \bullet]$, and the reduction implied by the latter item was performed, then a reduction of γ to C should still be inappropriate, and the conflict on the newly reduced symbol A can be ignored. In other words if there is no derivation

$$S \stackrel{*}{\Rightarrow} \theta\gamma X\phi,$$

then there can be no derivation

$$S \stackrel{*}{\Rightarrow} \theta\gamma AX\phi \stackrel{*}{\Rightarrow} \theta\gamma X\phi.$$

If there were two such derivations, then there will always be some symbol X in the lookahead sets of both item K and item J, and a conflict on X would result. Thus the conflict on A can be ignored as it either represents no real conflict, or if there is a real conflict, it will be signaled by another lookahead symbol X.

The symbol X will always be visible since by Algorithm NPG_3, no reductions are made for which the lookahead symbol is invisible. □

Lemma 4.9.3: Shift-reduce conflicts on any invisible symbol can be resolved in favour of shifting. That is, if a state contains two items, one of the form $[B \rightarrow \alpha \bullet A\beta]$, and the other of the form $[C \rightarrow \gamma \bullet]$ with a lookahead set that includes A, where A is in V_I, then A can be deleted from the lookahead set of the second item.

Proof: The case where A directly produces ϵ is handled by Lemma 4.9.2 above, so we need to prove this lemma only for the case where A indirectly produces ϵ (that is $A \Rightarrow \psi \stackrel{+}{\Rightarrow} \epsilon$). The proof is quite similar to the proof of Lemma 4.9.2, but the decision to reduce the string ψ to A will not be made by state r, the state with the conflict. Rather it will be made by some state t on a path ψ from state r. Nevertheless, CLOSURE or step (2.4.4)'' will generate from item $I = [B \rightarrow \alpha \bullet A\beta]$, at least two items $[A \rightarrow \bullet\psi]$ and $[D \rightarrow \bullet]$, where ψ is in V_I^+, and D is in FIRST(ψ). Since ψ is composed solely of invisible symbols, D is also in LAST(A), and hence every symbol in FOLLOW(A) will also

be in FOLLOW(D). Before ψ can be reduced to A, and A pushed onto the lookahead stack, the symbol D must be reduced from ϵ. Before the reduction of D, some symbol X, part of further right context must appear on the lookahead stack. If the reduction of γ to C was inappropriate on lookahead X before the reduction of D and A it should still be inappropriate after. In other words if there is no derivation

$$S \stackrel{*}{\Rightarrow} \theta \gamma X \phi,$$

then there can be no derivation

$$S \stackrel{*}{\Rightarrow} \theta \gamma A X \phi \stackrel{*}{\Rightarrow} \theta \gamma X \phi.$$

If there were two such derivations, then there will always be some symbol X in the lookahead sets of both item K and item J, and a conflict on X would result. Thus the conflict on A can be ignored as it either represents no real conflict, or if there is a real conflict, it will be signaled by some other lookahead symbol X.

The symbol X will always be visible since by Algorithm NPG_3, no reductions are made for which the lookahead symbol is invisible. □

Lemma 4.9.4: Reduce-reduce conflicts on invisible symbols that directly produce ϵ can be ignored in Algorithm NPG_2.

Proof: Suppose that the conflict is on the lookahead symbol A. Assume there is no shift-reduce conflict on A, since if there is Lemma 4.9.2 indicates that the correct action is to **shift**, and A should be deleted from the lookahead sets of the **reduce** actions thus eliminating the conflict. If there is no shift-item $[B \rightarrow \alpha \bullet A \beta]$ then Lemma 4.7.4 indicates that A cannot actually appear on the lookahead stack for this state, and hence A can be deleted from the lookahead sets of the two conflicting complete items. □

Lemma 4.9.5: Reduce-reduce conflicts on any invisible symbol can be ignored in Algorithm NPG_2.

Proof: Again we can assume that if there is a reduce-reduce conflict on the invisible symbol A in some state r, there is no shift-reduce conflict on the same symbol. If there were such a conflict then the correct action would be to **shift**, and the symbol A should be deleted from the lookahead sets of the conflicting **reduce** actions.

Invisible symbols can be ranked by the depth of parse tree they need to produce ϵ. Invisible symbols that directly produce ϵ, those symbols in the set $\{A \mid A \Rightarrow \epsilon\}$, are given a rank of 1. Other invisible symbols are given the highest rank of any symbol they produce plus one. (In an unambiguous grammar, invisible symbols can have no recursion, and therefore their rank must be finite. This is easily shown using Lemma 4.9.1 and its corollary.) By the corollary to Theorem 4.2, we can delete a symbol A from the lookahead sets of complete items, unless some expanded state has an expansion item of the form $[A \rightarrow \bullet \alpha]$ and there is no shift item $[B \rightarrow \beta \bullet A \gamma]$ in the same state. Such an item can only exist if there is a conflict on some symbol in α. By induction on the rank of an invisible symbol, using Lemmas 4.9.2 and 3.4 as a basis, and the previous statement as the induction step, it can be shown that all reduce-reduce conflicts on invisible symbols that indirectly produce ϵ, can be ignored. □

The proof of Lemma 4.8.1 required that the grammar contain no invisible symbols. An examination of that proof will show, however, that it is sufficient that the lookahead sets contain no invisible symbols.

Lemma 4.9.6: Invisible symbols can be deleted from FR_i, the sets of fully-reduce lookahead symbols.

Proof: Symbols in FR_i are those that may not be shifted and reduced by expansion items to resolve conflicts. If no invisible symbols appear in L_i or F_i then no conflicts can arise on invisible symbols. (All conflicts involve a **reduce** action.) Since no conflicts can arise on invisible symbols, no shift items will be created to resolve conflicts on invisible symbols. Since no such shift items are

created, there is no need to keep invisible symbols in FR_i to block such actions.

It is still possible, however, that state expansion items whose shift symbol is invisible will be created by step (2.4.3)'' of Algorithm NPG_2 to resolve conflicts on some visible symbol X. This step may create items of the form $[B \rightarrow \bullet C\delta]$ where C is invisible, and FRONT(δ) contains the conflicting lookahead symbol X. In this case however, if $C \Rightarrow \epsilon$ directly, then step (2.4.4.1)'' will always generate a complete item of the form $[r, 0] = [C \rightarrow \bullet]$ whose lookahead set L_r will contain all symbols in FOLLOW(C) $\cap$ V_V. If C would have been in some FR_i to block the shifting of C, then since C is nullable, FR_i will also contain some visible symbol Y in FOLLOW(C) and there will be an unresolved reduce-reduce conflict between L_i and L_r. Therefore, invisible symbols can be deleted from the sets FR_i without allowing grammars that would shift and reduce invisible lookahead symbols that should not be reduced.

If $C \stackrel{*}{\Rightarrow} \epsilon$ indirectly, rather than directly, then step (2.4.4.1)'' will generate an item $[r, 0] = [D \rightarrow \bullet]$, such that D is in FIRST(C). The lookahead set of that new item, L_r, will contain all the symbols in FOLLOW(D) $\cap$ V_V. Since C is invisible, FOLLOW(D) $\supseteq$ FOLLOW(C). Therefore as with the case in the preceding paragraph, there will be a conflict between L_i and L_r, and the conflict that would have arisen on C can be safely ignored. □

Theorem 4.3: Every SLR(1) consistent grammar will be accepted by Algorithm NPG_3.

Proof: This theorem follows from Lemmas 4.7.1, 4.8.1, 4.9.3, 4.9.5, and 4.9.6. □

4.10. Algorithm NPG_3': A Variations on Algorithm NPG_3

Algorithms NPG_1 through NPG_3 have the property that they do not reject any grammars except by leaving conflicts in the generated parser. This approach has the advantage that no special treatment is needed for rejected grammars, but it also has two disadvantages. The first disadvantage is that sometimes work is done generating expansion items for states when it is obvious that the conflicts cannot be eliminated. The second disadvantage is that understanding how the algorithm works can be clouded by the complex mechanisms by which conflicts are left in the parser to ensure rejection of the grammar.

If one allows a parser generator to have a mechanism for rejecting grammars other than by leaving conflicts in the parser, and allows some grammars to be rejected when resolving conflicts becomes too complex, then shorter simpler parser generators can be designed. Such a parser generator has been presented by Salomon and Cormack.[49,50] That algorithm is reproduced here for comparison purposes, and is called Algorithm NPG_3'. To present that algorithm, an additional function definition is needed.

Definition: $HIDDEN(A) = \{X \mid X \text{ is in } V_V, A \Rightarrow \alpha X \beta, \ \alpha \neq \epsilon \text{ and } \alpha \stackrel{+}{\Rightarrow} \epsilon\}$

Algorithm NPG_3' is presented in the informal style of the original paper in which it appeared.[49,50] The alternate style may even help clarify the operation of preceding parser generators presented here.

Algorithm NPG_3' avoids the complexity of steps (2.4.3)'' and (2.4.4)'' by rejecting all grammars that require recursive addition of state expansion items. Almost all such grammars lead to unresolvable conflicts anyway. There is, however, a small class of grammars accepted by Algorithm NPG_3, but rejected by Algorithm NPG_3'. An example of a grammar that falls into this class is:

Loop until all states completed.
    Compute next SLR state q for grammar.
    To each complete item $I_i = [A \rightarrow \alpha \bullet]$ attach a lookahead set $L_i = \text{FOLLOW}(A) \cap V_V$ and a fully-reduced lookahead set $FR_i = \text{FR\_FOLLOW}(A) \cap V_V$.
    If state 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 FR_i or X is in $HIDDEN(B)$ 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 q
                    end for
                end if
            end for
            If *resolved* then
                remove X from L_i
            else
                reject grammar
            end if
        end for
    end if
end loop.

Algorithm NPG_3'

$$S \rightarrow BC \mid ED$$
$$C \rightarrow Aec$$
$$D \rightarrow Aed$$
$$A \rightarrow \epsilon \mid a$$
$$B \rightarrow b$$
$$E \rightarrow b$$

The language described by this grammar has four sentences: *bec*, *bed*, *baec*, and *baed*. In the SLR(1) parser generated form this grammar there will be a conflict on whether to reduce the initial b of each sentence to B or E. Algorithm NPG_3 will insert state-expansion items $[C \rightarrow \bullet Aec]$ and $[D \rightarrow \bullet Aec]$ to reduce multiple lookahead symbols to C or D thus resolving the conflict on the reduction of B and E. In contrast Algorithm NPG_3' will simply reject this grammar.

The properties of the grammars accepted by Algorithm NPG_3 but rejected by Algorithm NPG_3' are more restrictive than the above example would indicate. The right parts of the expansion item added must begin with the same nullable symbol, and that symbol must directly produce ϵ. Nevertheless, this kind of grammar may occur in the description of real programming languages where white space (represented by the symbol A) is inserted at the start of tokens instead of at the end as in the grammar of Appendix E.

4.11. Reducing Lookahead-Set Size for Complete Items

Tai points out in Section 8 of his paper[55] that state expansion generates some useless elements in lookahead sets for complete items, and he comments on a possible solution to this problem. With the corrections of Algorithms NPG_2 and NPG_3, many unnecessary elements are added to the lookahead sets of unexpanded states too. In this section, methods of reducing the number of these useless elements are presented.

The corollary to Theorem 4.2 tells us that some nonterminals will never be left on the lookahead stack long enough to serve as lookahead for a **reduce** action. A nonterminal is pushed onto the lookahead stack by a **reduce** action. After reducing a string α to a nonterminal A, the parser always returns to a state r that contains an initial item of the form $[A \rightarrow \bullet \alpha]$. This will be the state that initiated the shifting of the symbol α_1. If this state also contains an item of the form $[B \rightarrow \beta \bullet A \gamma]$ then the symbol A will be immediately shifted from the lookahead stack, and will never serve as the lookahead symbol for any complete item.†

In an SLR(1) consistent state, all initial items (items with the dot at the start of the right part) are generated by the CLOSURE operation from other items in the same state.* As a result every initial item of the form $[A \rightarrow \bullet \alpha]$ in an unexpanded state will have a parent item of the form $[B \rightarrow \beta \bullet A \gamma]$ in the same state. In an expanded state, on the other hand, some initial items are generated from the lookahead sets of complete items, and have no parent item to shift on the reduced symbol.

Algorithm DUR_1: Delete Useless Reduce Actions

Let *RWOS* (reduce without shift) be the set of symbols that can be reduced without an immediate shift.

$$RWOS = \{A \mid \text{some state } s \text{ contains an item of the form } [A \rightarrow \bullet \alpha], \text{ but } s \text{ does not contain an item of the form } [B \rightarrow \beta \bullet A \gamma]\}$$

A simple method of reducing the number of nonterminal entries in the lookahead sets of complete items is to delete all nonterminals that are not in *RWOS* from the lookahead sets of complete items. □

Continued analysis of the parsing process shows that the size of lookahead sets can be reduced even further. Suppose a reduction pushes a symbol A in *RWOS* onto the lookahead stack and returns to a state r. That instance of A can be used as a lookahead symbol for further **reduce** actions only so long as it is not shifted onto the state stack. Once it is shifted, it cannot reappear on the lookahead stack unless it is reduced to some other symbol or a different instance of A. If, after a reduction pushes a symbol A onto the lookahead stack, the symbols on the state stack form a string γ, then any state that can use that instance of A as a lookahead will have symbols δ on the state stack that were reduced from γ and possibly ϵ. Neither A nor any further right context could have been involved in the reduction to δ. Since δ was reduced from γ and possibly ϵ, there must exist a derivation $\delta \stackrel{*}{\Rightarrow} \gamma$.

The path taken from the start state to the current state will determine the string currently on the state stack. Each shift transition on the path contributes one symbol of the string. Let γ be a string that can be on the state stack when the symbol A is reduced and the parser returns to a state r that contains no shift transition for A. There can be a large and possibly infinite number of such strings γ for any given state r. Let Γ be the set of all such strings γ for all states that do not shift A after reduction. Let Δ be the set of strings that can be on the state stack when a reduction is indicated by a complete item I in state r. If $\delta \stackrel{*}{\Rightarrow} \gamma$ for some string δ in Δ, and some string γ in Γ, then we know

† This statement assumes that the parser has no remaining shift-reduce conflicts. For parse tables that may be put to use even though they have shift-reduce conflicts, we must consider how the parser resolves such conflicts. If it resolves them by shifting, then the statement is still true, otherwise it is also necessary to check that no complete items use A as a lookahead symbol.

* The only exception is the item $[S' \rightarrow \bullet \vdash S \dashv]$ which can be treated specially.

that item I in state r may need the symbol A in its lookahead set.

Applying this test to each complete item of the parsing automaton will determine which nonterminals may be on top of the lookahead stack for each complete item of each state. Such an analysis may be quite complex, so we propose a simpler but less stringent test. If the nonterminal A is to be useful as a lookahead symbol for a complete item $[C \rightarrow \theta D \bullet]$ then there must be some expanded state that does not shift A but contains a complete item $[B \rightarrow \theta \bullet]$ whose lookahead set includes A and either

1) B is in LAST(D), or
2) D is nullable and there is some symbol E such that $E \stackrel{*}{\Rightarrow} \phi BD$ for some ϕ.

To apply this simplified test, define the function TAIL_FOLLOW as:

$$\text{TAIL\_FOLLOW}(X) = \{Y \mid A \stackrel{*}{\Rightarrow} \alpha XY \text{ for some } A\}.$$

Also define the function PNTLA(B) (mnemonic for *Possible NonTerminal LookAhead*) that gives nonterminals that could possibly be used as lookahead symbols for complete items $[C \rightarrow \gamma B \bullet]$, or $B \rightarrow \gamma$ if $\gamma \stackrel{*}{\Rightarrow} \epsilon$. PNTLA can be computed as follows.

Algorithm *CPNTLA*: Compute PNTLA

```
(1)  For each expanded state r:
(2)     For each item [A → • α] in r:
(3)        For each complete item [i,n_i] = [B → β •] in r:
(4)           If A is in L_i then:

(5)              For each C such that B is in LAST(C):
(6)                 Add A to the set PNTLA(C)).

(7)              For each nullable symbol C in TAIL_FOLLOW(B):
(8)                 Add A to the set PNTLA(C).
```

A second algorithm for eliminating useless **reduce** actions from the parser action table can now be given.

Algorithm *DUR*$_2$: Delete Useless Reduce Actions

```
(1)  For each state q:
(2)     RWOS_q = ∅.
(3)     If q is an expanded state then:

(4)        For each item [A → • α] in q:
(5)           Add A to RWOS_q.

(6)        For each item [B → β • A γ] in q:
(7)           Remove A from RWOS_q if present.

(8)     For each complete item [j,n_j] = [B → β •] in q:
(9)        k = |β|.
(10)       For each nonterminal C such that f(q, C) = reduce j:
(11)          If C is not in RWOS_q then:
(12)             If β = ε and C is not in PNTLA(B) then:
(13)                Change f(q, C) from reduce j to error.
(14)             If β ≠ ε and C is not in PNTLA(β_k) then:
(15)                Change f(q, C) from reduce j to error.
```

In this algorithm, the set $RWOS_q$ is a subset of $RWOS$ and contains those elements of $RWOS$ contributed by state q. In other words it contains those symbols that can be pushed onto the lookahead stack by a reduction that returns to state q, but for which state q contains no **shift** actions. No symbols in $RWOS_q$ should be deleted from the lookahead set of any **reduce** action in state q, since any

of those symbols could legitimately appear as lookahead symbols. Step (11) ensures that such symbols are not deleted.

PNTLA can be computed incrementally after the expansion of a state, but the elimination of useless **reduce** entries from f must be performed after the construction of the parser action function f is complete. The set FOLLOW(A) is a superset of TAIL_FOLLOW(A) and hence can be used in place of the latter set at the expense of possibly increasing the number of useless **reduce** actions that remain.

Algorithms DUR_1, *CPNTLA* and DUR_2 were implemented and applied to a scannerless NSLR(1) parser for ISO Pascal. Algorithm DUR_1, deleted 89.6% of **reduce** actions on nonterminal lookahead symbols, consequently deleting 41% of the total number of table entries. Algorithm DUR_2 (implemented to use FOLLOW in place of TAIL_FOLLOW) deleted 97.4% of **reduce** actions on nonterminal lookahead symbols, decreasing the total table size by 45%. Higher complexity algorithms for deleting useless **reduce** actions can be devised, but with diminishing returns. Since the remaining **reduce** actions on nonterminal lookaheads for our sample parser constitute only 2% of the total table size, and this figure could never be reduced to 0% in a parser with expanded states, the rewards for developing better algorithms would be small. There may, however, exist other grammars that could benefit more substantially from a further improved algorithm.

Not all parser implementations benefit from the use of Algorithms DUR_1 or DUR_2. These algorithms do not actually reduce the number of entries in a parser action table, but rather replace **reduce** actions by **error** actions. Thus only parsers that use sparse table storage techniques that do not store **error** entries will in fact be smaller. Nevertheless, in some implementations of parsers the smaller number of **reduce** actions may actually speed up parser execution. This case would arise when a searching algorithm is used to locate an entry in a sparse table. For an example of such a parser implementation see Aho & Ullman[4] Section 6.8, page 233.

4.12. Algorithm NPG$_4$: Complete Corrected NSLR(1) Parser Generator

(1) Initially, let s_0 = CLOSURE({[0, 0]}) and $Q = \{s_0\}$ with s_0 "unmarked."

(2) For each unmarked state s in Q, mark it by performing the following steps:

(2.1) Compute the SLR(1) lookahead sets:

(2.1.1) For each i, $0 \leq i \leq p$, let

$$L_i = \begin{cases} \text{FOLLOW}(A) \cap V_V & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

be the *simple 1-lookahead set* associated with the reductions on P_i.

(2.1.2) Let $L = \{Y \mid s \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow \alpha, \text{ and } \alpha_{j+1} = Y\}$ be the *simple 1-lookahead set* associated with all shift transitions from s.

(2.1.3) If L, L_0, L_1, "...," and L_p are pairwise disjoint, then s is SLR(1) consistent, so go to step (2.7). [Steps (2.2) to (2.6) comprise the *state expansion* part of the algorithm performed only on inconsistent states.]

(2.2)'' For each i, $0 \leq i \leq p$,

$$FR_i = \begin{cases} \text{FR\_FOLLOW}(A) \cap V_V & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

$$F_i = \begin{cases} \text{FOLLOW}(A) \cap V_V & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow \alpha \\ \varnothing & \text{otherwise} \end{cases}$$

(2.3)'' Let $R = \{Y \mid s$ contains $[i, j]$, $j < n_i$, $P_i = A \rightarrow \alpha$ and $\alpha_{j+1} = Y\}$ be the set of shift symbols of s.

(2.4)'' For each i, $0 \leq i \leq p$,

(2.4.1)'' Compute the largest subset R_i of F_i that can distinguish the reduction by P_i: $R_i = \{Y$ in $F_i \mid Y$ is in neither R nor F_j, where $0 \leq j \leq p$ and $j \neq i\}$.

(2.4.2)'' Let $L_i = FR_i \cup R_i$ be the *NSLR(1)-lookahead set* associated with the reductions on P_i.

(2.4.3)'' Form the set I_i of new items where $I_i = \{[q, 0] \mid P_q = B \rightarrow \beta$, B is in F_i, $\beta \neq \epsilon$, and for some X in FRONT(β), X is in $F_i - L_i\}$.

(2.4.4)'' For each unmarked item $[q, 0] = [B \rightarrow \bullet\beta]$ in I_i, such that $C = \beta_1$ is in V_N, and there is a rule $P_r = C \rightarrow \gamma$, where $\gamma \stackrel{*}{\Rightarrow} \epsilon$:

(2.4.4.1)'' Add $[r, 0] = [C \rightarrow \bullet\gamma]$ to I_i.

(2.4.4.2)'' Mark the item $[q, 0]$.

(2.4.5)'' Add the set I_i to s.

(2.4.6) Update PNTLA as in Algorithm *CPNTLA* steps (2) to (8).

(2.5)' Let $L = \{Y \mid s$ contains $[i, j]$, $j < n_i$, $P_i = A \rightarrow \alpha$, and $\alpha_{j+1} = Y\}$ be the *NSLR(1)-lookahead set* associated with all shift transitions from s.

(2.6)' If L, L_0, L_1, "...," and L_p are not pairwise disjoint, then abort (thus G is not noncanonical SLR(1)).

(2.7) For each Y in L, set $f(s, Y) =$ **shift**. If $L_0 = \{\dashv\}$, then set $f(s, \dashv) =$ **accept**. For each Y in L_i, $1 \leq i \leq p$, set $f(s, Y) =$ **reduce** i. For each Y in $V \cup \{\dashv\}$ but not in $L \cup L_0 \cup L_1 \cup \cdots \cup L_p$, set $f(s, Y) =$ **error**.

(2.8) For each Y in L, compute GOTO(s, Y) as follows:

$$\text{GOTO}(s, Y) = \text{CLOSURE}(\{[i, j+1] \mid s \text{ contains } [i, j],\ j < n_i,\ P_i = A \rightarrow \alpha, \text{ and } \alpha_{j+1} = Y\}).$$

If GOTO(s, Y) is not already in Q, then add it to Q as an unmarked state. Set $g(s, Y) = \text{GOTO}(s, Y)$.

(2.9) For each Y in V but not in L, set $g(s, Y) =$ **error**.

(3) Delete useless **reduce** actions from F using Algorithm DUR_2. □

4.13. Summary

This chapter describes in detail the NSLR(1) parser and parser generation method as described by Tai. An error in Tai's construction is identified. Before the error can be corrected, Tai's handling of epsilon productions is improved. In addition, this improvement is essential to parsing character-level grammars, the subject of this thesis. The improved parser generator, called Algorithm NPG_1 is shown to accept a proper superset of the grammars accepted by the original algorithm.

After improving the handling of epsilon productions, a correction for the error in Tai's Method is proposed. The parser generation method employing this correction is called Algorithm NPG_2. It is proved that the correction does not introduce any new conflicts so long as the grammar contains no invisible symbols. The necessity of a case by case analysis in that proof is demonstrated by the fact that the method does fail for the special case of invisible symbols.

We next prove that invisible symbols in lookahead sets can be ignored for the parser generation method proposed. The parser generator that ignores invisible symbols in the lookahead set is called Algorithm NPG_3. A significantly simplified version of NPG_3 is also presented that is only slightly less powerful.

Finally, the correction proposed to Tai's error generates many useless symbols in the lookahead sets of parser actions. These useless entries may double the storage requirements for the parse tables if sparse table storage methods are in use. As a result a method is proposed for eliminating most of these useless symbols. The method has achieved at least 97% efficiency on real character-level grammars. The completed parser generation algorithm is presented in its entirety and called NPG_4.

Chapter 5

Restrictive Rules for Context-Free Grammars

5.1. Introduction

In this chapter we present a detailed description of the restrictive rules proposed for context-free grammars. The description of each of the proposed rules is given in three parts.

(1) A formal definition is given of the meaning of the new rules.

(2) Algorithms are given for transforming context-free grammars containing instances of these rules (called *restricted context-free grammars—RCFG's*) into pure context-free grammars. In this way the closure properties of these new rules on context-free grammars are established.

(3) Algorithms are given for generating parsers for restricted CFG's.

5.1.1. Restricted Context-Free Grammars

▸ A *restricted context-free grammar*, an RCFG, is a quadruple $G = (V, P, R, S)$ where:

$V = V_N \cup V_T$ = The set of symbols in the grammar.

V_N = The set of nonterminal symbols.

V_T = The set of terminal symbols.

P = The set of productions.

$R = R_E \cup R_{AR}$ = The set of restrictive rules.

R_E = The set of exclusion rules.

R_{AR} = The set of adjacency-restriction rules.

S = A member of V_N, is the start symbol.

As with ordinary context-free grammars, productions, the elements of P, take the form $A \rightarrow \alpha$, where A is in V_N and α is a string in V^*. The forms of the exclusion rules, R_E, and the adjacency restriction rules, R_{AR}, are presented in detail below in Sections 5.2 and 5.3, respectively. It can be seen that every restricted context-free grammar $G = (V, P, R, S)$, has a corresponding unrestricted CFG $G^U = (V_N, V_T, P, S)$, which is the same a G but without the restrictive rules. The definitions of the restrictive rules given below imply that $L(G) \subseteq L(G^U)$.

The purpose of the restrictive rules is to contradict the productions in some way, and to override the language defined by the unrestricted grammar. Thus the restrictive rules are applied *after* the productions to eliminate certain derivations specified by the productions.

5.2. The Exclusion Rule

The form of an exclusion rule is the same as the form of an ordinary CFG production except that the operator " $\rightarrow$ " is replaced by the operator " $\not\rightarrow$ ". A sample exclusion rule would be:

$$A \not\rightarrow \alpha$$

The left part of an exclusion rule is in V_N and the right part is in V^*. In BNF grammars that use " ::= " or " = " as the production operator, the symbols " $::\neq$ " or " $\neq$ ", respectively, can be used for

the exclusion operator. If a grammar must be represented in the ASCII character set, then the character sequences "-X->", " ::# ", or " # " can be used as the exclusion operator.

An exclusion rule such as

$$identifier \nrightarrow begin$$

can be read as "*identifier* does not generate *begin*", or "*identifier* excludes *begin*". It indicates that the language generated by the left part (*identifier*) should not include the language generated by the right part (*begin*). Such a rule could be used with a general description for *identifier*, to exclude the generation of specific sentential forms by *identifier*. The following grammar fragment illustrates this use of the exclusion rule.

identifier → *id*
id → *letter* | *id letter* | *id digit*
begin → "b" "e" "g" "i" "n"
end → "e" "n" "d"
if → "i" "f"
then → "t" "h" "e" "n"
else → "e" "l" "s" "e"
...
identifier ↛ *begin* | *end* | if | *then* | *else*

Without the added exclusion rule, the language described by the symbol *identifier* includes those described by the symbols *begin*, *end*, *if*, *then*, and *else*. If those keywords are supposed to be reserved, then this is not the desired effect. The given exclusion rules remove the keywords from the set of valid identifiers. The above example also illustrates that, just as with ordinary CFG rules, multiple exclusion rules with the same left part can be joined by combining the right parts separated by OR bars "|".

The language excluded by an exclusion rule is the one defined by productions only, not by other exclusion rules, and thus multiple exclusion rules are independent. For example, if a grammar G contains the two exclusion rules $A \nrightarrow B$ and $B \nrightarrow C$ then the language generated by A excludes the language generated by B as described by the productions for B, disregarding the exclusion rule for B.

5.2.1. Formal Description of the Exclusion Rule

Consider an RCFG $G = (V, P, R, S)$ such that R_E contains the exclusion rule $A \nrightarrow \alpha$. If $L_{G^U}(\alpha)$ is the sublanguage generated by the string α according to the unrestricted grammar, then

$$L_G(A) = L_{G^U}(A) \cap \overline{L_{G^U}(\alpha)}.$$

In other words, the exclusion rule forbids derivations of the form $S \stackrel{*}{\Rightarrow} \beta A \gamma \stackrel{*}{\Rightarrow} \beta \delta \gamma$ for all δ in $L(\alpha)$. In the case of multiple exclusion rules with the same left part such as $A \nrightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ the meaning is

$$L_G(A) = L_{G^U}(A) \cap \overline{L_{G^U}(\alpha_1)} \cap \overline{L_{G^U}(\alpha_2)} \cap \cdots \cap \overline{L_{G^U}(\alpha_n)}$$

Since the language generated by symbols in the left part of exclusion rules is defined in terms of the unrestricted grammar, multiple exclusion rules with different left parts do not interact.

5.2.2. Closure Properties of the Exclusion Rule

Consider an RCFG G' defined by adding the exclusion rule $A \nrightarrow \alpha$ to a CFG G. The exclusion rule specifies that $L_{G'}(A)$, the language generated by the symbol A according to the grammar G', should be the language $L_G(A) \cap \overline{L_G(\alpha)}$. Since context-free languages are not closed under complementation,

and it is known to be undecidable whether the intersection of two CFL's is also a CFL, context-free grammars are not closed under exclusion. (Note that the language defined by a grammar using an exclusion rule, is still well defined, even if the language is not context-free.)

If we wish an RCFG to produce a context-free language, we must place restrictions on the use of exclusion rules. CFL's are closed under intersection with a regular set, and regular sets are closed under complementation, so if $L_G(\alpha)$ were restricted to being a regular set, then $L(G)$ would be context-free. With this restriction a grammar transformation can be given for constructing a CFG G'' from an RCFG G'. If the sole purpose of the exclusion rule were to solve the reserved-keyword problem then restricting $L_G(\alpha)$—the description of the reserved keywords —to being a finite set, would still provide enough power to write a language specification.

Algorithm *TE*: Grammar Transformation for Eliminating Exclusion Rules

This algorithm makes use of several algorithms presented by Hopcroft and Ullman.[24] These algorithms are subscripted by the number of the theorem in which each appears. The algorithms used are:

$HU_{2.1}$ – page 22 – Convert an NFA without ϵ moves to a DFA.

$HU_{2.2}$ – page 26 – Convert an NFA with ϵ moves to an NFA without ϵ moves.

$HU_{3.2}$ – page 59 – Given a DFA Mthat recognizes a language $L(M)$ construct a DFA that recognizes the language $\overline{L(M)}$.

$HU_{5.3}$ – page 115 – Given a CFG G, construct a DPDA that recognizes $L(G)$.

$HU_{5.4}$ – page 116 – Given a DPDA M, construct a CFG G such that $L(G)$ is the language recognized by M.

$HU_{6.5}$ – page 135 – Given a DPDA M_1 and a DFA M_2 construct a DPDA M_3 which recognizes the language $L(M_1) \cap L(M_2)$.

$HU_{6.6b}$ – page 137 – Test if a CFL is finite.

$HU_{9.1}$ – page 218 – Given a left- or right-linear grammar G construct an NFA that recognizes $L(G)$.

Consider a context-free grammar $G = (V_N, V_T, P, S)$ with the added exclusion rule $A \nrightarrow \alpha$. The CFG G, with this added rule, becomes the RCFG $G' = (V, P', R, S)$. The language $L(G')$ could be given by a pure CFG $G'' = (V_N'', V_T, P'', S)$ constructed using the following steps:

(1) Initially P'' contains all rules in P except those whose left part is A.

(2) Ensure that $L_G(\alpha)$, is regular, and if so build a DFA that recognizes that language. It is undecidable, in the general case, whether a CFL is regular. (See, for instance, Hopcroft and Ullman,[24] page 281.) There are, however, some CFL's that are readily recognizable as regular.

i) It is decidable if a CFL is finite (by algorithm $HU_{6.6}$), and all finite sets are regular. Building a DFA to recognize a finite set is simple.

ii) If the grammar for $L_G(\alpha)$ is left linear, or right linear, then it is regular. Construct a DFA from this grammar using algorithms $HU_{9.1}$, $HU_{2.2}$, and $HU_{2.1}$.

iii) If $L_G(\alpha)$ is a deterministic CFL then it is decidable whether it is regular. Stearns[53] and Valiant[56] give decision algorithms for the regularity of a DPDA, and an algorithm (though not a practical one) for constructing a finite-state machine from a DPDA. Unfortunately it is not decidable in the general case whether a CFL is deterministic.

If an DFA for recognizing $L_G(\alpha)$ cannot be constructed, then the RCFG grammar is rejected.

(3) Compute a grammar $G_A = (V_{AN}, V_{AT}, P_A, A)$ for the language $L(G_A) = L_G(A) \cap \overline{L_G(\alpha)}$. Since $L_G(\alpha)$ is regular, $\overline{L_G(\alpha)}$ is also regular. This step can be done using algorithms $HU_{3.2}$, $HU_{5.3}$, $HU_{6.5}$, and $HU_{5.4}$.

(4) Add the productions P_A to P'', and the symbols in V_{AN} to the set V_N''.

(5) Remove useless productions.

A grammar containing multiple exclusion rules with the same left part, such as $A \nrightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$, can be processed by replacing those rules with the rule $A \nrightarrow B$, and adding the productions $B \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ to P before applying Algorithm *TE*. A grammar containings multiple exclusion rules with different left parts can be processed by applying Algorithm *TE* repeatedly, using the output grammar of each application as the input grammar for the next application. This can be done since multiple exclusion rules are independent as described in Section 5.2.1 above. It is important to note, however, that wherever Algorithm *TE* refers to $L_G(\alpha)$, it refers to the language generated by the right part of the exclusion rule according to the original grammar as it existed before any applications of the algorithm.

5.2.3. Generating LR Parsers for Grammars Containing Exclusion Rules

Our construction algorithm builds the NSLR parser for the unrestricted ambiguous grammar, and uses the exclusion rules to eliminate actions that would parse the excluded sublanguages. The method consists of two modifications to the parser generator, and three tests for membership in the class of grammars that are handled correctly.

Definition: The set DESCEND(A) contains the immediate descendants of A.

$$\text{DESCEND}(A) = \{X \mid A \Rightarrow \alpha X \beta \text{ for some } \alpha, \beta\}$$

Definition: The set DESCEND*(A) contains all symbols that can appear in any sentential form generated by A.

$$\text{DESCEND}^*(A) = \{X \mid A \stackrel{*}{\Rightarrow} \alpha X \beta \text{ for some } \alpha, \beta\}$$

Definition: The set DESCEND$^+$(A) contains all symbols that can appear in any sentential form generated by A, except A itself.

$$\text{DESCEND}^+(A) = \text{DESCEND}^*(A) - \{A\}$$

Definition: The set U_DESCEND(A), the unique descendants of A, contains all symbols, including A itself, that can appear only in sentential forms generated by A.

$$\text{U\_DESCEND}(A) = \{X \mid \text{Every derivation } S \stackrel{*}{\Rightarrow} \alpha\beta X\gamma\delta \text{ generates the same parse tree as some derivation } S \stackrel{*}{\Rightarrow} \alpha A\delta \stackrel{*}{\Rightarrow} \alpha\beta X\gamma\delta\}.$$

For example, in the grammar of Appendix D, U_DESCEND(*Identifier*) contains *Identifier*, *IdentifierFragment* and *Letter*.

Definition: The uniquely-LAST function, U_LAST(A), is the set of symbols that only appear as the last symbol of sentential forms generated by A, and includes A itself.

$$\text{U\_LAST}(A) = \{Y \mid A \stackrel{*}{\Rightarrow} \beta Y \text{ for some } \beta, \text{ and every derivation } S \stackrel{*}{\Rightarrow} \alpha\beta Y\gamma \stackrel{*}{\Rightarrow} z \text{ generates the same parse tree as some derivation } S \stackrel{*}{\Rightarrow} \alpha A\gamma \stackrel{*}{\Rightarrow} \alpha\beta Y\gamma \stackrel{*}{\Rightarrow} z\}$$

Figure 5.1 gives an algorithm for computing U_LAST(A) and U_DESCEND(A).

$$\text{END\_SYM}(B, X) = \begin{cases} \textbf{true} & \text{if } B \Rightarrow \alpha X \\ \textbf{false} & \text{otherwise} \end{cases}$$

Compute the transitive closure END_SYM$^+$.

$$\text{ALT\_DESCEND}(B, Y) = \begin{cases} \textbf{true} & \text{for } Y \text{ in DESCEND}(B) \\ & \quad \text{and } B \neq A \\ \textbf{false} & \text{otherwise} \end{cases}$$

Compute the transitive closure ALT_DESCEND$^+$.

Initially U_LAST(A) = $\{X \mid \text{END\_SYM}^+(A, X)\}$,
and U_DESCEND(A) = DESCEND$^+$(A).

```
For each B in DESCEND*(A):
    For each rule B → αY:
        For each symbol X in α:
            U_LAST(A) := U_LAST(A) − DESCEND*(X).
        end for
    end for
end for

For each B such that ALT_DESCEND+(S, B) is
        true, where S is the start symbol:
    Remove B from U_LAST(A).
    Remove B from U_DESCEND(A).
end for

Add A to U_LAST(A) and U_DESCEND(A).
```

Figure 5.1. An algorithm for computing U_LAST(A) and U_DESCEND(A).

Algorithm *PGSE*: Parser Generation for Simple Exclusion Grammars

To implement the processing of exclusion rules, the following modifications are made, and tests added, to an LR parser generator:

Modification EM1: For each rule $\mathbf{E} \twoheadrightarrow \mathbf{F}$, reduce-reduce conflicts between two items $I_i = [C \rightarrow \gamma \bullet]$ and $I_j = [D \rightarrow \delta \bullet]$, where C is in U_DESCEND(**E**), and D is in U_LAST(**F**), are resolved in favour of the second item, the reduction to D. Specifically, if L_i and L_j are the lookahead sets for items I_i and I_j respectively, then for each X in L_j remove X from L_i. In addition, if FR_i and FR_j are the fully-reduced follow sets for items I_i and I_j respectively, then for each Y in FR_j remove Y from FR_i. The conflict resolution is made before each SLR state is tested for consistency. If state expansion added ϵ-items then the conflict resolution should also be applied again after state expansion.

The effect of Modification EM1 is to prefer chained reductions that will yield **F** rather than **E**. We will call this *inhibiting a reduction to* **E**.

Modification EM2: For each exclusion rule $\mathbf{E} \twoheadrightarrow \mathbf{F}$, add a specially marked rule $\mathbf{E} \dot{\rightarrow} \mathbf{F}$. If the grammar already contains a rule $\mathbf{E} \rightarrow \mathbf{F}$, that rule should be specially marked, rather than adding a new rule. Without the addition of the specially marked rules, the parser may not recognize that a string in $L(\mathbf{F})$ has occurred where only a string in $L(\mathbf{E})$ is expected according to the original grammar. These added rules will be handled like any other production by the parser generator except that no

reduce actions should be produced due to complete items of the form $[\mathbf{E} \dot{\rightarrow} \mathbf{F} \bullet]$, and the rules should be treated as if they do not exist during the computation of the sets DESCEND, DESCEND*, DESCEND$^+$, U_DESCEND, and U_LAST. (A parser generator that treats the marked rules like any other production during the computation of the above five sets will accept a smaller class of grammars, but will still accept most of the grammars of interest.)

Test ET1: This test enforces the restriction on the grammar that the right part of each exclusion rule must be a single nonterminal. In other words, the right part may not be a string or a terminal.

Test ET2: For Modification EM1 to be correct we must ensure that for any two parses of **E** and **F**, either the parses are distinct, or that they start simultaneously. This test consists of two steps.

(a) Ensure that no recognition of **E** starts during a recognition of **F**. There must be no states in the parser that contain an item of the form $[A \rightarrow \alpha \bullet \mathbf{E}\beta]$ and (i) an item of the form $[B \rightarrow \gamma X \bullet Y\delta]$ where B is in DESCEND*(**F**), or (ii) an item of the form $[B \rightarrow \gamma \bullet]$ where B is in DESCEND*(**F**) but not in U_LAST(**F**).

(b) Ensure that no recognition of **F** starts during a recognition of **E**. The conditions are the same as for step (a) above but exchange **E** and **F**.

Test ET3: For each exclusion rule $\mathbf{E} \twoheadrightarrow\!\!\!\!\!\!\times\ \mathbf{F}$, ensure that whenever a recognition of **F** has completed, a concurrent recognition of **E** also ends. This test can be done by ensuring that FOLLOW(**F**) $\cap$ DESCEND$^+$(**E**) = $\varnothing$.

Tests ET1 to ET3 should be applied after all state processing is complete, including state expansion, if any.

The above method accepts only a restricted class of grammars, which we call SE (*simple exclusion*) grammars, and which we have found adequate to describe programming languages. The adjective *simple* is used to describe this class of grammars because membership can be tested by examining each state of the generated parser individually without regard to the transitions between states. This property is shared by SLR grammars.

The above implementation can be simplified if we note that for most grammars of interest, U_LAST(**E**) = {**E**} and U_LAST(**F**) = {**F**}. As a result, an implementation that omits the computation of U_LAST(**E**) and U_LAST(**F**) and uses {**E**} and {**F**}, respectively, in their place will still have adequate power for the intended applications of the the exclusion rule.

5.2.4. Correctness of Algorithm *PGSE*

In the discussion that follows we will assume that the exclusion rule being implemented is $\mathbf{E} \twoheadrightarrow\!\!\!\!\!\!\times\ \mathbf{F}$, and the bold symbols **E** and **F** will be used without further comment.

Modification EM2 ensures that for every derivation $S \stackrel{*}{\Rightarrow} \alpha\mathbf{E}\beta$, there will also be a derivation $S \stackrel{*}{\Rightarrow} \alpha\mathbf{F}\beta$. If however, the second derivation requires the use of one of the specially marked productions generated by Modification EM2, then parsing of the second sentential form will be blocked, since the specially-marked productions cannot generate **reduce** actions.

If there is a derivation $S \stackrel{*}{\Rightarrow} \alpha\mathbf{E}\beta$, and a derivation $S \stackrel{*}{\Rightarrow} \alpha\mathbf{F}\beta \stackrel{*}{\Rightarrow} \alpha x\beta$, that does not require the use of one of the specially marked rules, then there are two possibilities: (a) the string x is not in $L(\mathbf{E})$, or (b) the string x is in $L(\mathbf{E})$. Case (a) is of no further interest since the rule $\mathbf{E} \twoheadrightarrow\!\!\!\!\!\!\times\ \mathbf{F}$ does not affect such derivations. In case (b), there would be two derivations of the string $\alpha x\beta$, and the exclusion rule would disallow the one of the form $S \stackrel{*}{\Rightarrow} \alpha\mathbf{E}\beta \stackrel{*}{\Rightarrow} \alpha x\beta$. The two possible derivations of $\alpha x\beta$ would cause a reduce-reduce conflict in the generated parser on the reductions $\mathbf{E} \stackrel{*}{\Rightarrow} x$, and $\mathbf{F} \stackrel{*}{\Rightarrow} x$. The remainder of this proof will show that many such conflicts will be resolved in favour of the second reduction. We need not prove that all **E**-**F** conflicts are resolved; we need only show that all the conflicts that are resolved really are **E**-**F** conflicts, and are resolved correctly. If unresolved conflicts remain, the generated parser is rejected. The intention is that the set of grammars, the simple-exclusion grammars, for which all conflicts are resolved, should be a useful set.

An outline of the remainder of the proof is as follows. We prove that whenever our method favours a reduction **reduce** j to a symbol in U_LAST(**F**) over a reduction **reduce** i to a symbol in U_DESCEND(**E**),

(i) reduction **reduce** j ends the recognition of a sentence x generated by **F**,
(ii) reduction **reduce** i ends the recognition of a sentence y generated by **E**, and
(iii) the recognition of x and y started in the same state.

Since the recognitions of x and y start in the same states and end in the same states, therefore $x = y$, and the reduction to **E** should indeed have been inhibited in order that $\mathbf{E} \nrightarrow x$. Item (iii) is proved by showing that

(a) if a recognition of **E** is initiated while a recognition of **F** is in progress, or
(b) if a recognition of **F** is initiated while a recognition of **E** is in progress,

then the grammar will be rejected by Test ET2.

5.2.4.1. Proof for SLR Parser Generators

First we prove that Algorithm *PGSE* works for an SLR parser generator. In the next subsection, we prove it for the more difficult case of a NSLR parser generators.

To make the discussion that follows simpler, we assume that the state stack of an LR parser contains both states attained by shift transitions, and the symbols shifted by those transitions. Since the latter information is not necessary for the correct implementation of LR parsers it is customarily omitted from actual implementations.

Lemma 5.1: In an LR parser, if the top $n+1$ elements of the state stack are the sequence of states $\hat{q} = \hat{q}_0 \cdots \hat{q}_n$, with $\hat{q}_n$ being the topmost, and the top n symbols on the state stack are the string $\alpha = \alpha_1 \cdots \alpha_n$, and state $\hat{q}_n$ specifies a **reduce** action on the production $A \rightarrow \alpha$, then state $\hat{q}_0$ must contain an item of the form $[B \rightarrow \beta \bullet A\gamma]$ for some B, β, and γ.

Proof: This is a fundamental property of LR parsers and a full proof is not given here. We can see, however, that for such a reduction to be specified by state $\hat{q}_n$, states $\hat{q}_i$ for i in the range $0 \leq i \leq n$ must each contain an item of the form $[A \rightarrow \delta \bullet \varsigma]$, where $\delta\varsigma = \alpha$ and $|\delta| = i$. Thus state $\hat{q}_0$ contains an item of the form $[A \rightarrow \bullet \alpha]$ and such items can only be generated by item-set closure on items of the form $[B \rightarrow \beta \bullet A\gamma]$. □

Definition: ▸ A *final recognition configuration* Π for a symbol A in an LR parsing automaton M is a configuration of M such that the symbols on the top of the state stack can be reduced to A by a series of **reduce** actions alone. Thus a final recognition configuration Π is a quintuple $\Pi = (A, M, \hat{q}, \alpha, a)$ where A is the symbol to be recognized, M is the parsing automaton being considered, $\hat{q}$ is a sequence of $n+1$ states on top of the state stack, α is the string of the top n symbols on the state stack that are to be reduced to A, and a is the lookahead symbol. The state $\hat{q}_n$ and the symbol α_n are at the top of the state stack.

When M is in the final recognition configuration $\Pi = (A, M, \hat{q}, \alpha, a)$ it is prepared to perform the sequence of reductions that corresponds to the derivation $A \Rightarrow \beta \Rightarrow \gamma \Rightarrow \cdots \Rightarrow \alpha$ where each string is generated by expanding the final symbol of the previous string, if it is a non terminal. (This is not the same as a right-most derivation where the right-most nonterminal is expanded.) Since a parser in a final recognition configuration can be transformed from having α on the top of the state stack to having A there by **reduce** actions alone, at each step of the transformation it is only a suffix of the state stack that is replaced.

Another property of final recognition configurations of the form $\Pi = (A, M, \hat{q}, \alpha, a)$ is that each state $\hat{q}_i$ for $0 \leq i < n$ is connected to state $\hat{q}_{i+1}$ by a shift transition on the symbol α_{i+1}.

Lemma 5.2: State $\hat{q}_0$ of a final recognition configuration $\Pi = (A, M, \hat{q}, \alpha, a)$ must have an item of the form $[B \rightarrow \beta \bullet A\gamma]$.

Proof: This is easily shown by repeated application of Lemma 5.1 on $\hat{q}$. □

Lemma 5.3: When Modification EM1 inhibits a reduction to a symbol C on lookahead symbol b in state r of a parser M, it is known that when M is in state r it will always be in a final recognition configuration for **F**.

Proof: When Modification EM1 inhibits such a reduction on lookahead symbol b, then state r contains an item $I_j = [D \rightarrow \delta \bullet]$, such that D is in U_LAST(**F**), and the lookahead set L_j of I_j contains the symbol b. Since D can appear only at the end of sentential forms generated by **F** it is guaranteed that **F** can be reduced from the top of the state stack without shifting on any more symbols. In addition, since L_j = T_FOLLOW(D), and since T_FOLLOW(D) = T_FOLLOW(**F**) for all symbols D in U_LAST(**F**), therefore b is a valid lookahead symbol for a reduction to **F**. Therefore the parser would be in a final recognition configuration for **F**. □

Lemma 5.4: When Modification EM1 inhibits a reduction to C on lookahead symbol b in a parser state r, then if the reduction were not inhibited, whenever the parser would be in state r it would always be in a final recognition configuration for the symbol **E**.

Proof: When Modification EM1 inhibits a reduction to C in a parser state r, then state r contains an item $[C \rightarrow \gamma \bullet]$, such that C is in U_DESCEND(**E**). Since members of U_DESCEND(**E**) can only appear in sentential forms generated by **E**, the string currently on top of the state stack must form at least part of a string that is eventually reduced to **E**. Modification EM1 inhibits reductions to C on lookahead symbols in the set L_j for the item $[D \rightarrow \delta \bullet]$. Test ET3 ensures that

$$\text{FOLLOW}(\mathbf{F}) \cap \text{DESCEND}^{+}(\mathbf{E}) = \varnothing.$$

Since D is in U_LAST(**F**), therefore FOLLOW(D) = FOLLOW(**F**). Since

$$L_j = \text{T\_FOLLOW}(D) \subseteq \text{FOLLOW}(D),$$

therefore

$$L_j \cap \text{DESCEND}^{+}(\mathbf{E}) = \varnothing.$$

Therefore the lookahead symbol could not possibly be part of any string reduced to **E**, and the string currently on top of the state stack must form all of the string to be reduced to **E**.

In addition we know that b is in T_FOLLOW(C). Since C is in U_DESCEND(**E**), therefore

$$\text{T\_FOLLOW}(C) \subseteq \text{T\_FOLLOW}(\mathbf{E}) \cup \text{DESCEND}^{*}(\mathbf{E}).$$

But by Test ET3 we know that b is not in DESCEND*(**E**). Therefore b must be in T_FOLLOW(**E**) and would be a valid lookahead symbol for a reduction to **E**. Therefore the parser must be in a final recognition configuration for **E**. □

Lemma 5.5: In a final recognition configuration $\Pi = (A, M, \hat{q}, \alpha, a)$ of length n every state $\hat{q}_i$ for $0 < i < n$ has an item of the form $[B \rightarrow \beta X \bullet Y\gamma]$ for some B in DESCEND*(A).

Proof: The state sequence $\hat{q}$ is produced by repeated application of the CLOSURE and the GOTO functions starting with an item of the form $[C \rightarrow \delta \bullet A\varsigma]$. In order that α_j for $0 < j \leq n$ could be shifted onto the state stack, all kernel items of state $\hat{q}_j$ must have the form $J = [B' \rightarrow \phi\alpha_j \bullet \psi]$, for some B', ϕ, and ψ. Furthermore, at least one of these items B' must be in DESCEND*(A), otherwise α_j could not participate in a reduction to A. If for all such items $\psi = \epsilon$ then j must equal n, that is state $\hat{q}_j$ must be the last state of $\hat{q}$, since in that case there would be no way for CLOSURE($\hat{q}_j$) and GOTO($\hat{q}_j$, ψ_1) to produce an item of the form J in state $\hat{q}_{j+1}$. Since $\psi \neq \epsilon$ for at least one item in all states $\hat{q}_i$ the lemma is satisfied with $B' = B$. □

Theorem 5.1: In parsers that pass all of the tests ET1 through ET3, if Modification EM1 of Algorithm *PGSE* inhibits the reduction of a string α to A on lookahead a in a state r of a parsing automaton M in favour of a reduction of β to B, then whenever M is in state r it would have been in a final recognition configuration $\Pi = (\mathbf{E}, M, \hat{q}, \gamma, a)$ of length n with $r = \hat{q}_n$ and that final

recognition configuration will be inhibited in favour of a final recognition configuration $\Psi = (\mathbf{F}, M, \hat{q}, \gamma, a)$. That is M will be in a final recognition configuration for $\mathbf{F}$ with an identical state sequence $\hat{q}$, reduction string γ, and lookahead symbol a as in Π.

Proof: Lemma 5.4 specifies that when M is in state r it must be in a final recognition configuration $\Pi = (\mathbf{E}, M, \hat{q}, \gamma, a)$ of length n with $\hat{q}_n = r$. Lemma 5.3 specifies that when M is in state r it must be in a final recognition configuration $\Omega = (\mathbf{F}, M, \hat{s}, \delta, a)$ of length m with $\hat{s}_m = r$. We must show that $\hat{q} = \hat{s}$, and $\gamma = \delta$.

There are three possibilities: $n = m$, $n < m$, or $n > m$.

Case 1: $n = m$. In this case since $\hat{q}$ and $\hat{s}$ must both be on the state stack at the same time, $\hat{q} = \hat{s}$. Since in an LR parsing automaton all shift transitions to a particular state must be on the same symbol, therefore $\gamma = \delta$, and the theorem is proved.

Case 2: $n < m$. In this case state $\hat{q}_0 = \hat{s}_{m-n}$. By Lemma 5.2, $\hat{q}_0$ contains an item $[C \rightarrow \theta \bullet \mathbf{E}\varsigma]$, and by Lemma 5.5 $\hat{s}_{m-n}$, and hence $\hat{q}_0$, must have an item $[D \rightarrow \phi X \bullet Y\psi]$ for some D in DESCEND*(**F**). But Test ET2(a) rejects parsers with two such items in the same state. Therefore this case is impossible in an accepted parser.

Case 3: $n > m$. An argument similar to the one used for Case 2 above will show that this case is also impossible in accepted parsers.

Therefore the lemma holds. □

5.2.4.2. Proof for Noncanonical SLR Parser Generators

The proof for *noncanonical* SLR parsers is complicated by the fact that state expansion can introduce into a state an item $[C \rightarrow \bullet \alpha]$ even though there is no item $[B \rightarrow \mu \bullet C'\varsigma]$ such that C is in FIRST(C'). This property means that lemmas 5.3, 5.4, and 5.5 do not always hold. We can however develop modified versions of these lemmas that allow the proof of Theorem 5.1 for noncanonical SLR parsers. Lemma 5.6 substitutes for lemmas 5.3 and 5.4, and Lemma 5.7 substitutes for 5.5.

Lemma 5.6: When Modification EM1 inhibits a reduction to a symbol C on lookahead symbol b in state r of a parser M, the language accepted by the parser will not change unless M is in a final recognition configuration for both $\mathbf{E}$ and $\mathbf{F}$ when in state r.

Proof: Since C is in U_DESCEND(**E**), inhibiting a reduction to C cannot change the language accepted by the parser unless C is part of a string that is later reduced to $\mathbf{E}$. Since b is not in DESCEND*(E), C must be at the end of such a string, and hence M must be in a final recognition configuration for E. Due to the productions added by Modification EM2, when M is in a final recognition configuration for $\mathbf{E}$ it must also be in a final recognition configuration for $\mathbf{F}$. □

Lemma 5.7: In a final recognition configuration $\Pi = (A, M, \hat{q}, \alpha, a)$ of length n every state $\hat{q}_i$ for $0 < i < n$ has an item of the form $[B \rightarrow \beta X \bullet Y\gamma]$, or an item of the form $[B \rightarrow \delta \bullet]$ for some B in DESCEND*(A).

Proof: The proof is similar to the proof for Lemma 5.5, except that items can be generated by CLOSURE, GOTO, or state expansion. The proof of Lemma 5.5 still holds for showing that there must be an item in state $\hat{q}_j$ of the form $J = [B' \rightarrow \phi\alpha_j \bullet \psi]$ where B' is in DESCEND*(A). But it is possible to have another item of that form in state $\hat{q}_{j+1}$ for $j < n$, even if $\psi = \epsilon$, by the application of state expansion on $\hat{q}_j$ after the application of CLOSURE($\hat{q}_j$). The item of the form J satisfies the lemma by setting $B' = B$, and either $\phi\alpha_j = \beta X$ and $\psi = Y\gamma$ if $\psi \neq \epsilon$, or $\phi\alpha_j = \delta$ if $\psi \neq \epsilon$. □

Theorem 5.1 can be proved for noncanonical SLR parsers too if its wording is changed slightly. Rather than saying that if Modification EM1 inhibits a reduction in a state r then when parser M enters state r it *must* be in a final recognition configuration for $\mathbf{E}$ and $\mathbf{F}$, the theorem should say that the inhibited reduction cannot affect the outcome of the parse unless M actually is in a final recognition configuration for $\mathbf{E}$ and $\mathbf{F}$. This wording is less clear, but works equally well for SLR and NSLR parsers.

To prove the reworded Theorem 5.1, Lemma 5.6 can be used in place of lemmas 5.3 and 5.4, and since Test ET2 test for both forms of the items mentioned in Lemma 5.7, Lemma 5.7 can be used in place of Lemma 5.5. Since the proof is virtually identical, it is not repeated here.

5.3. The Adjacency-Restriction Rule

An adjacency-restriction rule takes the form

$$W \not- X$$

The symbol " $\not-$ ", a dash with a slash through it, is intended to convey the meaning *may not be adjacent to*. If the grammar is to be represented using the ASCII character set, the sequence of characters "-/-" can be used to represent the adjacency-restriction delimiter.

If the rule $W \not- X$ is inserted into a CFG $G = (V_N, V_T, P, S)$, the effect is to disallow all derivations of the form

$$S \stackrel{*}{\Rightarrow} \alpha W X \gamma \stackrel{*}{\Rightarrow} z$$

and all derivations that would produce the same parse tree as a derivation of the above form. The intended use of an adjacency-restriction rule is to specify that in a derivation $S \stackrel{*}{\Rightarrow} \alpha W \beta X \gamma \stackrel{*}{\Rightarrow} z$, the string β may not derive ϵ. The above definition holds equally well even when $W \stackrel{*}{\Rightarrow} \epsilon$ or $X \stackrel{*}{\Rightarrow} \epsilon$.

In RCFG's for real programming languages, it is likely that many symbols would have similar adjacency restrictions. To shorten such grammars we allow that the left and right parts of an adjacency restriction rule may be strings of symbols. For example, the rule $\alpha \not- \beta$ means that $\alpha_i \not- \beta_j$ for all pairs i and j such that $1 \leq i \leq |\alpha|$, and $1 \leq j \leq |\beta|$. Note that the order of application of multiple adjacency restrictions has no effect on the language being defined.

5.3.1. Sample Usages of Adjacency Restrictions

To illustrate the use of the adjacency-restriction rule, two examples, drawn from actual grammars for programming languages, are presented. The first shows how to use adjacency-restriction to solve the longest-match ambiguity for identifiers and keywords. The second example shows how to resolve the dangling-else ambiguity discussed in detail in Section 2.4.1.

As was mentioned in Chapter 1, the standard grammars for Pascal are ambiguous on how to parse the code fragment

```
BEGINWORKEND;
```

It could be parsed as either a simple statement that invokes a procedure called `BEGINWORKEND`, or as a compound statement containing only a call to the procedure `WORK`. As is shown in the following grammar fragment, a simple set of adjacency restrictions on the symbol that generates identifiers, and those that generate the reserved words, can resolve the ambiguity.

ID → *id*
id → *letter* | *id letter*
letter → *a* | *b* | *c* | · · · | *z*
white → ϵ | " "
BEGIN → *b e g i n*
END → *e n d*
· · ·
a → "a" | "A"
b → "b" | "B"
c → "c" | "C"
· · ·
ID BEGIN END · · · $\not\dashv$ *ID BEGIN END* · · ·

In this example the ellipses represent the rules and symbols needed to complete the grammar for the remainder of the alphabet, and the remainder of the reserved words. With the above adjacency restrictions, the ambiguity of the sample code segment is eliminated. It can no longer be parsed as a compound statement, since that would require a parse in which the symbol *BEGIN* is adjacent to the symbol *ID* (which produces the string `WORK`), and in which the symbol *ID* is adjacent to the symbol *END*. It can now be parsed only as a single identifier.

The second sample usage of adjacency restrictions presented here shows how adjacency restrictions can be used to resolve the dangling-else ambiguity. Consider the following grammar fragment:

Statement → *Labels ULStatement*

Labels → ϵ
| *Integer* ":" *Labels*

ULStatement → ϵ
| *Id* ":=" *Expr*
| *VarExpr* ":=" *Expr*
| *Id*
| *Id* "(" *ExprList* ")"
| **BEGIN** *StatList* **END**
| *IfThen*
| *IfThenElse*
| **CASE** *Expr* **OF** *CaseBody* **END**
| **WHILE** *Expr* **DO** *Statement*
| **REPEAT** *StatList* **UNTIL** *Expr*
| **FOR** *Id* ":=" *Expr UpDown Expr* **DO** *Statement*
| **WITH** *Var***DO** *Statement*
| GOTO *Integer*

IfThen → **IF** *Expr* **THEN** *Statement*

IfThenElse → **IF** *Expr* **THEN** *Statement* **ELSE** *Statement*

IfThen $\dashv$ **ELSE**

In this grammar fragment, an else clause must always be combined with the immediately preceding un-elsed-if clause, since otherwise the parse would contain the symbol *IfThen* immediately adjacent to the symbol **ELSE**. The advantages of this method of resolving the dangling-else ambiguity over the usual method of grammar rewriting are that it is shorter, simpler, and avoids repetitive and risky rule duplication. It also more clearly identifies the ambiguity being resolved rather than relying on the deductive powers of the reader.

In the interest of brevity, the treatment of white space has been omitted here, but the grammar in Appendix D contains a sample resolution of the dangling-else ambiguity in its entirety, including the treatment of white space.

5.3.2. Closure Properties of Adjacency Restrictions

In order to demonstrate that context-free languages are closed under adjacency restrictions, we present a grammar transformation that converts a context-free grammar with added adjacency restrictions into a pure CFG with no adjacency restriction that generates the same language. Formally, given an RCFG $G' = (V, P, R, S)$ with $R_E = \varnothing$, and $R_{AR} = \{W \dashv X\}$, the transformation produces a CFG $G'' = (V_N'', V_T'', P'', S)$ such that $L(G'') = L(G')$. It can be seen that G' is constructed from its corresponding unrestricted CFG $G = (V_N, V_T, P, S)$ by adding the restriction $W \dashv X$.

All new symbols created by this transformation are superscripted and subscripted versions of symbols in G. ▸ The terminology used here is that, for the adjacency restriction $W \dashv X$, any new symbol is made up of up to three parts: a *basic part* (the original symbol with no superscript or subscript), a possible *subscript part* of W or $\overline{W}$, and a possible *superscript part* of X or $\overline{X}$. When this

algorithm has completed the transformation, every sentence produced by a symbol subscripted by W must be derived through an intermediate string that ends in W, and every sentence produced by a symbol subscripted by $\overline{W}$ cannot be derived through and intermediate string that ends in W. That is to say, for every derivation $C_W \overset{*}{\Rightarrow} \alpha \overset{*}{\Rightarrow} y$, there must be some α such that $\alpha_{|\alpha|} = W$, and for every derivation $C_{\overline{W}} \overset{*}{\Rightarrow} \alpha \overset{*}{\Rightarrow} y$, $\alpha_{|\alpha|} \neq W$. Similarly, every sentence produced by a symbol superscripted by X must be derived through an intermediate string that begins with X, and every sentence produced by a symbol superscripted by $\overline{X}$ cannot be derived through and intermediate string that begins with X. That is to say, for every derivation $C^X \overset{*}{\Rightarrow} \alpha \overset{*}{\Rightarrow} y$, there must be some α such that $\alpha_1 = X$, and for every derivation $C^{\overline{X}} \overset{*}{\Rightarrow} \alpha \overset{*}{\Rightarrow} y$, $\alpha_1 \neq X$.

Algorithm *TAR*: Grammar Transformation for Eliminating Adjacency-Restriction Rules

Given an RCFG $G' = (V, P, R, S)$ constructed from a CFG $G = (V_N, V_T, P, S)$ by adding the restrictive rule $W \not\rightarrow X$; construct a CFG $G'' = (V_N'', V_T'', P'', S)$ such that $L(G'') = L(G')$.

(0) Initially $G'' = G$.

(1) Rewrite G'' to eliminate all ϵ-productions. An algorithm to do this is given by Hopcroft and Ullman,[24] in Section 4.4, page 90. If $S \underset{G}{\Rightarrow} \epsilon$ remember this fact, but temporarily disallow $S \underset{G''}{\Rightarrow} \epsilon$.

(2) For each symbol C in $\text{LAST}^{-1}(W)$:

  (2.1) For each occurrence of C in a production P_i'':

    (2.1.1) Replace P_i'' by two productions, one using C_W in place of C, and the other using $C_{\overline{W}}$.

(3) For each nonterminal C in $\text{FIRST}^{-1}(X) \cup \text{FIRST}^{-1}(X_W) \cup \text{FIRST}^{-1}(X_{\overline{W}})$:

  (3.1) For each occurrence of C in a production P_i'':

    (3.1.1) Replace P_i'' by two productions, one using C^X and the other using $C^{\overline{X}}$.

(4) For each new symbol S_B^A whose basic part is S (the original start symbol), add a new rule $S \rightarrow S_B^A$ to P''. S is still the start symbol for grammar G''.

(5) Delete all productions using symbols whose basic part is W and whose subscript part is $\overline{W}$. Similarly, delete all productions using symbols whose basic part is X and whose superscript part is $\overline{X}$. According to the meaning of the subscripts and superscripts, these symbols are self contradictory: a W that does not generate W and an X that does not generate X.

(6) Delete all productions of the form $C_W \rightarrow \alpha Z_{\overline{W}}$, $C_{\overline{W}} \rightarrow \alpha Z_W$, $C^X \rightarrow Z^{\overline{X}}\alpha$, or $C^{\overline{X}} \rightarrow Z^X\alpha$.

(7) Delete all productions of the form $C_W \rightarrow \alpha Y$ for Y not subscripted by W unless $C = W$. Also delete all productions $C^X \rightarrow Y\alpha$ for Y not superscripted by X unless $C = X$.

(8) Delete all productions $C \rightarrow \alpha Y_W Z^X \beta$. This is the main step of this algorithm; applying the principle of the adjacency-restriction rule.

(9) Remove all useless symbols and productions.

(10) If $S \underset{G}{\Rightarrow} \epsilon$ add the rule $S \rightarrow \epsilon$ to P''.

□

5.3.3. Sample Application of Algorithm *TAR*

To illustrate the application of Algorithm *TAR*, consider the RCFG $G_{5.1}$, and the resulting transformed CFG $G_{5.2}$.

$$
\begin{array}{lll}
S & \rightarrow & oper \mid ID \mid S\ white\ oper \mid S\ white\ ID \\
ID & \rightarrow & id \\
id & \rightarrow & letter \mid id\ letter \\
letter & \rightarrow & \text{"a"} \mid \text{"b"} \mid \text{"c"} \mid \cdots \mid \text{"z"} \\
white & \rightarrow & \epsilon \mid \text{" "} \\
oper & \rightarrow & \text{"*"} \mid \text{"/"} \mid \text{"+"} \mid \text{"–"}
\end{array}
$$

$$ID \dashv\!\!\!\!- ID$$

Grammar $G_{5.1}$: A sample RCFG.

$$
\begin{array}{lll}
S & \rightarrow & S_{ID}^{ID} \mid S_{\overline{ID}}^{ID} \mid {}_{\overline{ID}}^{\overline{ID}} \mid S_{\overline{ID}}^{\overline{ID}} \\
S_{ID}^{ID} & \rightarrow & ID_{ID}^{ID} \mid S_{ID}^{ID}\ white\ ID_{ID}^{ID} \\
 & & \mid S_{ID}^{ID}\ white\ ID_{ID}^{ID} \mid S_{ID}^{ID}\ ID_{ID}^{ID} \\
S_{\overline{ID}}^{ID} & \rightarrow & S_{ID}^{ID}\ white\ oper \mid S_{\overline{ID}}^{ID}\ white\ oper \\
 & & \mid S_{ID}^{ID}\ oper \mid S_{\overline{ID}}^{ID}\ oper \\
S_{\overline{ID}}^{\overline{ID}} & \rightarrow & S_{ID}^{\overline{ID}}\ white\ ID_{ID}^{ID} \mid S_{\overline{ID}}^{\overline{ID}}\ white\ ID_{ID}^{ID} \\
 & & \mid S_{\overline{ID}}^{\overline{ID}}\ ID_{ID}^{ID} \\
S_{\overline{ID}}^{\overline{ID}} & \rightarrow & oper \mid S_{ID}^{\overline{ID}}\ white\ oper \\
 & & \mid S_{\overline{ID}}^{\overline{ID}}\ white\ oper \mid S_{\overline{ID}}^{\overline{ID}}\ oper \\
 & & \mid S_{\overline{ID}}^{\overline{ID}}\ oper \\
ID_{ID}^{ID} & \rightarrow & id \\
id & \rightarrow & letter \mid id\ letter \\
letter & \rightarrow & \text{"a"} \mid \text{"b"} \mid \text{"c"} \mid \cdots \mid \text{"z"} \\
white & \rightarrow & \text{" "} \\
oper & \rightarrow & \text{"*"} \mid \text{"/"} \mid \text{"+"} \mid \text{"–"}
\end{array}
$$

Grammar $G_{5.2}$: The result of applying Algorithm *TAR* to Grammar $G_{5.1}$.

5.3.4. Correctness of Algorithm *TAR*

We now show that for a CFG G'' generated from an RCFG G' by Algorithm *TAR*, $L(G'') = L(G')$. The algorithm operates on the context-free grammar G, which is the unrestricted version of the RCFG G'. The method of this section is to show that all sentences of $L(G)$ that can be derived without violating the restriction $W \dashv\!\!\!\!- X$ will be in $L(G'')$, and that no sentences of $L(G)$ that must be derived by violating $W \dashv\!\!\!\!- X$ are in $L(G'')$.

To discuss the transformation of grammar G by Algorithm *TAR*, the intermediate grammar produced by steps (0) to (i) is called grammar $G_{(i)}$. Grammar $G_{(3)}$, for example, is the intermediate grammar obtained after applying steps (0) through (3) to grammar G. Thus $G_{(0)} = G$, and

$G_{(10)} = G''$. We will now examine, in order, the steps of Algorithm *TAR* to determine the effect that each will have on the language described by the resulting grammar.

Since step (1) of Algorithm *TAR* is designed not to change the language produced by the grammar, and since steps (2) and (3) merely duplicate rules, therefore $L(G_{(3)}) = L(G_{(0)})$. Due to all the rule and symbol duplication, however, grammar $G_{(3)}$ is considerably more ambiguous than grammar G.

The start symbol S of the grammar is unique in that its usage is not confined to rule replacement; every derivation of a sentence in $L(G)$ begins with the start symbol S. In this sense, the symbol S can be seen to be the only symbol that is *externally accessible*. The purpose of step (4) is to handle this uniqueness by providing external access to the duplicated version of the original start symbol.

Steps (5) to (7) eliminate the redundancy of the intermediate grammar introduced by steps (2) to (4). To continue this discussion we need a notation to precisely describe derivations, so that different derivations that derive the same sentence can be distinguished.

Definition: ▸ A *derivation history* by a context-free grammar $G = (V_N, V_T, P, S)$ is a sequence of triples (α, p, s), where α is a sentential form in the derivation, p is the production that is applied to generate the sentential form of the derivation, and s is the index of the symbol in α that is to be expanded to produce the next sentential form in the derivation. By this description, the sentential form in the first triple will be the start symbol S, and the sentential form in the last triple will be in V_T^*.

Consider a derivation history Ω by grammar $G_{(1)}$,† consisting of a sequence of triples $\Omega_1, \Omega_2, \ldots, \Omega_n$, such that $\Omega_i = (\omega_i, p_i, s_i)$.‡ If the sentential forms ω have n_W occurrences of symbols in $\text{LAST}^{-1}(W)$, and n_X occurrences of symbols in $\text{FIRST}^{-1}(X)$, then there will be $2^{(n_W+n_X)}$ derivation histories by grammar $G_{(4)}$ that are identical except that each occurrence of a symbol Y in $\text{LAST}^{-1}(W) \cup \text{FIRST}^{-1}(X)$ in the sentential form and the production parts of each triple, will be replaced by a symbol whose basic part is Y, and which has a possible subscript of either W or $\overline{W}$ and a possible superscript of either X or $\overline{X}$.

Definition: ▸ Two derivation histories that are identical when every symbol in the derivation-string and production parts of their triples is replaced by the basic part of the symbol, are called *basically-identical* derivation histories.

Steps (5) to (7) eliminate the redundancy introduced by steps (2) to (4) and ensure that for each unique derivation history by grammar $G_{(1)}$ there will be exactly one basically-identical derivation history by grammar $G_{(7)}$. To verify this statement, consider a sentential form $\omega_h = \alpha Y \beta$ in a derivation history Ω by grammar $G_{(1)}$, where Y is in $\text{LAST}^{-1}(W)$. Let $\gamma_1, \gamma_2, \ldots, \gamma_m$ be the sequence of substrings of ω_h to ω_n derived from Y. Thus $\gamma_1 = Y$, and γ_m is in V_T^*.

Consider also the following two sequences generated using $G_{(7)}$:

(i) a derivation history Φ, that is basically identical to Ω, consisting of the triples (ϕ_i, q_i, s_i), and

(ii) the strings $\psi_1, \psi_2, \ldots, \psi_m$ that are substrings of ϕ_h to ϕ_n, that are derived from the symbol Z in ϕ_h that corresponds to Y in ω_h (ψ_i is basically identical to γ_i for $1 \leq i \leq m$, and Z is basically identical to Y).

We can say that there are two distinct cases: either some γ_i for $1 \leq i \leq m$ ends in W, or none do.

---

† Notice that we are using $G_{(1)}$, which has no ϵ-productions, rather than $G_{(0)}$.

‡ Note that this is a departure from the usual notation of this thesis. In this case ω_i is the sentential form of the i^{th} triple of the derivation history Ω, rather than the i^{th} symbol of the string ω.

Case 1: some γ_i ends in W. Since grammar $G_{(1)}$ through $G_{(7)}$ have no ϵ-productions, the last symbol of a sentential form is always the parent of the last symbol of the sentence that the sentential form ultimately generates. Step (5) ensures that all symbols whose basic part is W are subscripted by W. Step (6) ensures that only a symbol subscripted by W can produce a string that ends in a symbol subscripted by W. Therefore, if Z is to generate a string that ends in a symbol whose basic part is W, then Z must be a symbol subscripted by W.

Case 2: no γ_i ends in W. Since grammar $G_{(1)}$ through $G_{(7)}$ have no ϵ-productions, the last symbol of a sentential form is always the parent of the last symbol of the sentence that the sentential form ultimately generates. Steps (6) and (7) together ensure that any symbol U subscripted by W must produce a string that ends in a symbol subscripted by W, unless the basic part of U is W. The string ψ_m is made up entirely of terminals, and Step (2) subscripts only symbols in $\text{LAST}^{-1}(W)$, hence the only terminal symbols that could be subscripted by W would have a basic part of W. Therefore in this case Z may not be subscripted by W it must be subscripted by $\overline{W}$.

These two cases imply that for every symbol Y in $\text{LAST}^{-1}(W)$ used in a sentential form of a derivation history by $G_{(1)}$, the subscript of the corresponding symbol Z in a basically identical derivation history by $G_{(7)}$ is determined by whether or not Y actually generates a sting ending in W. A similar argument will show that for every symbol Y in $\text{FIRST}^{-1}(X)$ used in a sentential form of a derivation history by $G_{(1)}$, the subscript of the corresponding symbol Z in a basically identical derivation history by $G_{(7)}$ is determined by whether or not Y actually generates a string starting with X. Since basically-identical derivation histories by $G_{(1)}$ and $G_{(7)}$ can only differ by the subscripts used on symbols in $\text{LAST}^{-1}(W) \cup \text{FIRST}^{-1}(X)$ and since those subscripts are uniquely determined by the derivation by $G_{(1)}$, there can be at most one basically-identical derivation by $G_{(7)}$.

It must also be shown that for each derivation history Ω by $G_{(1)}$ there is always at least one basically-identical derivation history Φ by $G_{(7)}$. The method of constructing such a Φ from Ω is quite simple. For each symbol Y in $\text{LAST}^{-1}(W)$ in the sentential forms of Ω chose a basically identical symbol whose subscript is W or $\overline{W}$ according to whether Y actually does or does not, respectively, generate a string ending in W in Ω. For this strategy to be consistent it is necessary that no rules of the form $C_W \rightarrow \alpha Z_W$ or $C_{\overline{W}} \rightarrow \alpha Z_{\overline{W}}$ be deleted from G''. This condition is met by Algorithm *TAR*. A similar analysis shows that there is always a consistent replacement for symbols in $\text{FIRST}^{-1}(X)$. This replacement strategy guarantees that there is always at least one basically-identical derivation history by $G_{(7)}$ for each derivation history by $G_{(1)}$.

Step (8) does the actual application of the adjacency restriction. We have shown that after step (7), only symbols subscripted by W can generate strings ending in a symbol whose basic part is W, and all symbols subscripted by W must generate such a string. We have also shown that after step (7), only symbols superscripted by X can generate strings beginning in a symbol whose basic part is X, and all symbols superscripted by X must generate such a string. Therefore by deleting all rules in which a symbol subscripted by W is followed immediately by a symbol superscripted by X, all sentential forms that violate the adjacency restriction $W \not\dashv X$ will be inhibited from generation, and only such sentential forms will be inhibited.

Step (9) is included to point out that the preceding steps may result in grammars with useless symbols and productions. If the grammar is to undergo further processing, these should be removed.

5.3.5. Repeated Application of Algorithm *TAR*

Algorithm *TAR* does some symbol renaming, therefore when it is used on RCFG's with more than one adjacency restriction, the meaning of subsequent adjacency restrictions may be lost. If Algorithm *TAR* processes a grammar to eliminate a rule $W \not\dashv X$, then any subsequent rule $Y \not\dashv Z$ in which Y is in $\text{LAST}^{-1}(W)$ in the original grammar, must be augmented by the two rules $Y_W \not\dashv Z$ and $Y_{\overline{W}} \not\dashv Z$. Similar changes must be made if Z is in $\text{LAST}^{-1}(W)$ or if Y or Z are in $\text{FIRST}^{-1}(X)$.

5.3.6. Parser Generation for RCFG's Containing Adjacency Restrictions

One could generate parsers for RCFG's by applying Algorithm *TAR* to the grammar and then using standard parser generation methods for the resulting CFG's. Such a method, however, would not be very practical since, as was seen with grammar $G_{5.2}$, Algorithm *TAR* can significantly increase the number of symbols and rules in a grammar, and hence the size of the parsing automaton. We present a parser generation method that produces parsers for RCFG's that are the same size or smaller than parsers for the corresponding unrestricted CFG.

The principle of the proposed method is that, since the derivations allowed by the restricted grammar are a subset of the derivations allowed by the unrestricted grammar, a parser for the restricted language can be built by placing restrictions on parser for the unrestricted language. The restrictions consist principally of deleting **reduce** actions and **shift** actions from the unrestricted parser and testing the consistency of the resulting parser.

The proposed method is intended for use with SLR or NSLR parser generators, and handles only a limited class of RCFG's called SAR (simple adjacency-restriction) grammars. The adjective *simple* is used to describe this class of grammars because membership in the class can be tested by examining each state of the generated parser individually without regard to the transitions between the states. This property is shared by simple LR parsers.

When the method is applied to an SLR parser generator, the class of grammars accepted is called SAR-SLR, and when applied to an NSLR parser generator, the class is called SAR-NSLR. The method we propose can also be applied to LALR and full LR parser generators, but in those parser generators, extra information is available about lookahead sets that can be put to use in providing a more-powerful treatment of adjacency restrictions.

The proposed method of parser generation for SAR grammars consists of three modifications to an LR parser generator, and three consistency checks of the grammar and generated parser.

The modifications are:

ARM1– Delete symbols from the lookahead sets of complete items. This change forbids the parser from making certain reductions and thus either eliminates conflicts from a state (disambiguation) or causes the rejection of input sentences that are in the unrestricted language but not in the restricted language.

ARM2– Place restrictions on the shift items that can be introduced by the CLOSURE operation on states.

ARM3– Require that lookahead sets be tested for some **reduce** actions, even though they are in LR(0) consistent states.

The consistency tests are:

ART1– Reject grammars with adjacency restrictions in which the left part is a terminal symbol.

ART2– Apply a test to the parser to avoid complex interactions of restricted symbols and unrestricted symbols.

ART3– Reject parsers with certain forms of items added during noncanonical state expansion.

5.3.6.1. Mod. ARM1 – Pruning Lookahead Sets

Before describing how lookahead sets are pruned, a new adjacency-restricted follow function, AR_FOLLOW, must be defined. AR_FOLLOW(W) is similar to FOLLOW(W) except that it excludes all symbols that can only occur in derivations that violate one of the adjacency restrictions.

$$\begin{aligned}\text{AR\_FOLLOW}(W) = \{X \mid\ & B \Rightarrow \alpha_0 Y_0 \beta Z_0 \gamma_0 \text{ for some } B,\\ & \beta \stackrel{*}{\Rightarrow} \epsilon, \text{ and there exist derivations}\\ & \alpha_0 Y_0 \Rightarrow \alpha_1 Y_1 \Rightarrow \cdots \Rightarrow \alpha_n Y_n \text{ where } W{=}Y_n\\ & Z_0 \gamma_0 \Rightarrow Z_1 \gamma_1 \Rightarrow \cdots \Rightarrow Z_m \gamma_m \text{ where } X{=}Z_m\\ & \text{such that no rule } Y_i \dashv\!\!\!\!+ Z_j \text{ exists for any } i, j\}\end{aligned}$$

In this (and the next) definition we depart from the normal use of subscripted strings: the subscripted greek letters α_0, α_1, ..., α_n, and γ_0, γ_1, ..., γ_n represent strings in V^*. Our normal notation is that α_i is the i^{th} symbol of the string α in V^*. Similarly Y_i and Z_i represent single symbols in V.

We also define AR_FOLLOW the adjacency restricted FR_FOLLOW set.

$$\begin{aligned}\text{ARFR\_FOLLOW}(A) = \{X \mid\ & B \Rightarrow \alpha_0 Y_0 \beta X \gamma \text{ for some } B,\\ & \beta \stackrel{*}{\Rightarrow} \epsilon, \text{ and there exists a derivation}\\ & \alpha_0 Y_0 \Rightarrow \alpha_1 Y_1 \Rightarrow \cdots \Rightarrow \alpha_n Y_n \text{ where } A{=}Y_n\\ & \text{such that no rule } Y_i \dashv\!\!\!\!+ X \text{ exists for any } i\ \}\end{aligned}$$

Algorithm *ARF* shows how these adjacency-restricted FOLLOW sets can be computed for an RCFG $G = (V, P, R, S)$.

Algorithm *ARF*: Computing Adjacency-Restricted Follow sets.

(1) For each rule $P_i = A \rightarrow \alpha$ in P:

(1.1) For each j in the range $|\alpha| > j \geq 1$ in descending order, (notice that when $\alpha = \epsilon$ this loop is skipped):

(1.1.1) For each k in the range $j < k \leq |\alpha|$ in ascending order, stopping after processing the first non-nullable α_k:

(1.1.1.1) *ProcessAdjacent*$(\alpha_j, \alpha_k, \varnothing, \varnothing)$.

PROCEDURE *ProcessAdjacent*(W, X, AW, CAR)
Where:
W, X = A pair of adjacent symbols in that order.
AW = Set of ancestors of W in current expansion.
CAR = Collected adjacency restrictions,
= $\{Z \mid Y \dashv\!\!\!\!+ Z$ is in R_{AR} and Y is in $AW\}$.

(1) If W is **not** in AW then:

(1.1) $CAR' = CAR \cup \{Y \mid W \dashv\!\!\!\!+ Y$ is in $R_{AR}\}$.

(1.2) If X is **not** in CAR' then:

(1.2.1) Insert X into ARFR_FOLLOW(W).

(1.2.2) *ProcessSuccessor*$(W, X, \varnothing, CAR')$.

(1.2.3) For each Y such that $W \rightarrow \alpha Y$ is in P:

(1.2.3.1) *ProcessAdjacent*$(Y, X, AW \cup \{W\}, CAR')$.

PROCEDURE *ProcessSuccessor*(W, X, AX, CAR)
Where:
W, X, and CAR have the same meaning as in *ProcessAdjacent*.
AX = Set of ancestors of X in current expansion.

(1) If X is **not** in AX then:

(1.1) Insert X into AR_FOLLOW(W).

(1.2) For each Y such that $X \rightarrow Y\alpha$ is in P:

(1.2.1) If Y is not in CAR then:

(1.2.1.1) *ProcessSuccessor*(W, Y, $AX \cup \{X\}$, CAR).

Algorithm *ARF* is a straightforward algorithm, and as such is computationally intensive. It functions by generating a significant part of all possible sentential forms generatable by G. Specifically, it generates all possible pairs of symbols that could be adjacent in any sentential form of G. During the generation, all ancestors of each symbol are recorded and tested for a violation of the adjacency-restriction rules in R_{AR}. To ensure termination, it avoids cycles in an expansion, caused by left- or right-recursive rules in P, by terminating expansion when a symbol appears as its own ancestor.

Mod. ARM1 for SLR Parsers

In SLR parsers, the lookahead set for the complete item $A \rightarrow \alpha\bullet$ is T_FOLLOW(A). When adjacency restrictions are applied, the lookahead sets should be pruned to eliminate following symbols that can only occur in sentential forms generated by derivations that violate the adjacency restrictions. To produce the pruned lookahead sets, define the function ART_FOLLOW(Y) to be T_FOLLOW(Y) after applying adjacency restrictions.

$$\text{ART\_FOLLOW}(Y) = \{a \text{ in } V_T \cup \{\dashv\} \mid a \text{ is in AR\_FOLLOW}(Y)\}.$$

The function ART_FOLLOW should be used in place of T_FOLLOW in Algorithm PG_0 (the SLR(1) construction algorithm) for the determining lookahead sets.

Mod. ARM1 for NSLR Parsers

Algorithm NPG_4, the improved NSLR(1) parser generation algorithm presented in the previous chapter, uses two follow functions: FOLLOW and FR_FOLLOW. To prune the lookahead sets of an NSLR(1) parser to comply with adjacency restrictions, these follow functions should be replaced by their adjacency-restricted counterparts AR_FOLLOW and ARFR_FOLLOW respectively. (To handle exclusion rules as well as adjacency restrictions, references to FOLLOW in Test ET3 of Algorithm *PGSE* should also be replace by AR_FOLLOW.)

5.3.6.2. Mod. ARM2 – Inhibit the Generation of Some Shift Actions

The lookahead-set pruning done by Modification ARM1, above, inhibits **reduce** actions that would violate the specified adjacency restrictions. It is also desirable to inhibit certain **shift** actions. Inhibiting these **shift** actions does not change the language accepted by the parser, but it does have three desirable effects: (1) it may eliminate shift-reduce conflicts with the inhibited **shift** actions, (2) it may eliminate shift transitions out of a state and hence may eliminate states from the parsing automaton thus reducing the size of the parser, (3) typically the eliminated states have parsing conflicts which will also be eliminated. The undesirable **shift** actions are generated by item-set CLOSURE on parser states and a simple redefinition of that function will eliminate them.

The original definition of the item-set CLOSURE operation on parser states was as follows:

> For any set I of items, let CLOSURE(I) be defined as the smallest set satisfying the following properties: (1) every item in I is in CLOSURE(I), and (2) if $[A \rightarrow \beta\bullet B\gamma]$ is in CLOSURE(I) and $B \rightarrow \delta$ is a production, then the item $[B \rightarrow \bullet\delta]$ is in CLOSURE(I).

It is a property of LR parsers that for any parser state there is a unique symbol Y such that all kernel items take the form $C \rightarrow \theta Y\bullet\phi$. The definition of item-set CLOSURE should be modified by adding the restriction that for all items $[B \rightarrow \bullet\delta]$ added by CLOSURE, B and δ_1 must be in AR_FOLLOW(Y). Items added by state expansion must also observe this rule.

5.3.6.3. Mod. ARM3 – Constrain Implementation of Generated Parser

No reduction may be made to a symbol A that appears as the left part of an adjacency restriction without verifying that the current lookahead symbol is in AR_FOLLOW(A). Some parser implementations perform **reduce** actions without checking for a valid lookahead symbol, provided that the state was LR(0) consistent. This simplification is still possible provided that the **reduce** action does not reduce to a symbol that appears as the left part of an adjacency-restriction rule. If such unchecked reductions are permitted the reduction-blocking method of applying adjacency restrictions may be bypassed.

5.3.6.4. Test ART1 – Limit the Use of Terminals in Adjacency Restrictions

For the implementation given here, adjacency-restriction rules provided by the grammar writer may not have a left part that is a terminal. That is, there may be no rule $W \not\dashv X$ in R_{AR} such that W is in V_T. The modifications given here work principally by blocking reductions to a symbol that is the left part of an adjacency restriction when the right part of the same restriction appears as the lookahead symbol. Since a terminal requires no reduction to appear on the parse stack, its reduction cannot be blocked. Note that this restriction is not a great hardship, since a unit production can easily be added to the grammar, so that the left part of the adjacency restriction rule can be a nonterminal.

5.3.6.5. Test ART2 – Reject Grammars with Complex Adjacency Restrictions

Consider the following grammar:

$$S \rightarrow BC$$
$$B \rightarrow D \mid b \mid aBc$$
$$D \rightarrow bb$$
$$C \rightarrow E \mid c$$
$$E \rightarrow cc$$
$$D \not\dashv E$$

Since D may not be followed by E we would not want D reduced to B with a lookahead of c, since the c lookahead may be part of a reduction to E. But since B may be followed by c, and since D is in LAST(B), the algorithms given above will still put c in the set AR_FOLLOW(D), which may permit an undesired reduction.

To reject grammars with this problem ensure that for each adjacency restriction $W \not\dashv X$, and for each item $[i, n_i] = [W \rightarrow \alpha \bullet]$ with lookahead set L_i then $L_i \cap \text{FIRST}(X) = \varnothing$. Grammars that exhibit this kind of problem contain a production of the form $A \rightarrow \alpha WY\beta$, and an adjacency restriction $W \not\dashv X$ such that $\text{FIRST}(Y) \cap \text{FIRST}(X) \neq \varnothing$.

5.3.6.6. Test ART3 – Reject Parsers with Certain Forms of Noncanonical Items

If there is a rule $W \not\dashv X$ in R_{AR}, then any state with an item $[i, n_i] = [B \rightarrow \beta \bullet]$ with lookahead set L_i where B is in LAST(W) may not also contain an item added during noncanonical state expansion of the form $[A \rightarrow \bullet \alpha]$, such that A is in L_i and either (i) X is in FIRST(α), or (ii) $\alpha \stackrel{*}{\Rightarrow} \epsilon$. Such an item may allow the reduction of a X to an ancestor symbol. Since the item was added by state expansion, it would indicate the reduction of right context before left context. Thus the principal mode of preventing adjacencies, that of blocking the reduction of left context when forbidden right context appears in the lookahead could be bypassed.

Note that it is not correct to delete such items, since they are required for the correct parsing of some sentences in the restricted language. It is necessary to detect grammars that produce such states, report the problem, and reject the grammar.

5.3.7. The Correctness of Adjacency-Restricted Parser Generation

In this section, we prove the correctness of the parser generation method by proving two properties of the generated parser:

(1) when the above modifications and tests are applied to an SLR or NSLR parser, all sentences accepted by the parser satisfy the adjacency restrictions specified, and

(2) if the grammar and generated parser pass the specified tests, then all sentences in the specified language will be accepted.

5.3.7.1. Property (1): All Sentences Accepted Are Valid

We now show that all sentences accepted by the generated parser will satisfy the adjacency restriction. Consider an RCFG $G = (V, P, R, S)$ and an adjacency restriction $W \dashv\!\!\!\!\!- X$. The symbol W must be either a terminal or a nonterminal.

Case 1: W is a terminal. This case is eliminated by Test ART1.

Case 2: W is a nonterminal. In any parse that violates the adjacency restriction, some string α must be reduced to W. Due to Modification ARM3, all reductions are performed only with a valid lookahead symbol, and the reduction to W has a lookahead set of AR_FOLLOW(W). We know by Test ART2 that

$$\text{AR\_FOLLOW}(W) \cap \text{FIRST}(X) = \varnothing,$$

therefore no reduction of α to W can take place with X or a leading descendant of X as the lookahead symbol. As a result, in any parse that violates the adjacency restriction, either some string β must be reduced to X and then to some symbol in $\text{FIRST}^{-1}(X)$ before W is reduced, or some symbol D, intervening between W and X must be reduced from ϵ. Since an SLR parser produces only rightmost derivations, neither two cases are possible for an SLR parser. An NSLR parser, however, can produce non rightmost parses. Consider each of the above two subcases for an NSLR parser.

Subcase 2a: Some string β is reduced through X to some predecessor of X before α is reduced to W. For this to happen there must be a state with an item $I = [A \rightarrow \bullet\, \alpha]$, such that X is in FIRST(α), and item I must have been added by noncanonical state expansion. (All canonical items lead to rightmost derivations only.) The state must also have an item $[B \rightarrow \beta \bullet]$ where β is left context that is ends in W, or β is left context yet to be reduced to W. Such combination of items are forbidden by Test ART3(i).

Subcase 2b: Some symbol D, intervening between W and X must is reduced from ϵ. This can only happen if a state has an item $[D \rightarrow \bullet\, \delta]$ is added by noncanonical state expansion and $\delta \stackrel{*}{\Rightarrow} \epsilon$. The state must also have an item $[B \rightarrow \beta \bullet]$ where β is left context that is ends in W, or β is left context yet to be reduced to W. Such combinations of items are forbidden by Test ART3(ii).

5.3.7.2. Property (2): All Valid Sentences Are Accepted

We prove here that if the parser is conflict free, and passes the three tests given above, then it will accept all valid sentences of the language described by the grammar being implemented.

Tests ART1, and ART2 can only reject grammars, and hence do not change the language recognized by a parser generated from an accepted grammar. Similarly Test ART3 rejects parsers, and hence does not change the language recognized by an accepted parser. Modification ARM3 does not disallow any valid sentences, it only disallows occasional parsing short-cuts. Thus only modifications ARM1 and ARM2 could disallow valid parses.

Modification ARM1 can only remove symbols from the lookahead sets of **reduce** actions, since AR_FOLLOW(A) $\subseteq$ FOLLOW(A). By the definition of AR_FOLLOW, when Modification ARM1 removes a symbol Y from the lookahead set of a complete item $[B \rightarrow \beta \bullet]$, then there can be no valid derivation $S \stackrel{*}{\Rightarrow} \alpha B Y \gamma$. Hence Modification ARM1 cannot inhibit any valid parses.

The **shift** actions deleted by Modification ARM2 would always lead to invalid parses if ever executed. Any symbol not in AR_FOLLOW(Y), should not be shifted onto the parse stack on top of Y. □

5.3.8. Additional Parser Enhancements

The strategy for implementing adjacency restrictions described above yields parsers with adequate power for accepting complete grammars for modern programming languages such as Pascal. There are further enhancements that can be made to the this implementation that will allow the parser to accept a larger set of grammars, and may also make grammar writing easier.

5.3.8.1. Eliminating Test ART1

Test ART1 of the adjacency-restriction implementation strategy presented above, the test that forbids adjacency restrictions with a left part that is a terminal symbol, seems rather artificial. It is actually possible to modify the parser further so that this test is not needed. For the rule $a \dashv X$, the change would involve preventing the reductions of strings ending in a, and an additional test that would reject parsers in tricky cases involving ϵ-productions. These changes are fairly complex but as explained earlier would be of very limited value. As a result they are neither described here, nor are they implemented in the prototype parser generator.

5.3.8.2. Mod. ARM4 – Reducing Parser Rejections by Test ART2

It is possible to make an additional change to an SAR-NSLR(1) parser that will reduce the number of grammars rejected by Test ART2. Test ART2 will reject a grammar when there is a rule $A \dashv B$ in R_{AR}, there is a symbol C such that FIRST(C) $\cap$ FIRST(B) $\neq \varnothing$, and there is a valid derivation $S \stackrel{*}{\Rightarrow} \alpha AC\gamma$. In such a case, when a symbol Y in FIRST(C) $\cap$ FIRST(B) appears as the lookahead symbol on the lookahead stack, the parser cannot know whether to reduce some string β to A, since Y may be at the start of a string to be reduced to B, an illegal reduction, or it may be the start of a string to be reduced to C, a legal reduction.

The solution is to create a new kind of resolvable parser conflict. State-expansion items should be created to shift on the lookahead symbol Y in the hopes that it will be reduced to either B or C, and resolve the conflict. This can be done by making the following two modifications to Algorithm NPG_4:

1) In step 2.1.3, if there is a complete item $[i, n_i] = A \rightarrow \beta\bullet$, and some symbol Y in FIRST(C) $\cap$ FIRST(B) appears in L_i, consider the state to be SLR(1) inconsistent, and proceed to noncanonical state expansion.
2) After step (2.4.1)'', delete Y from R_i.

Even when these two changes have been made, Test ART2 will still be required. If Y appears in FR_i then it will be reintroduced into L_i by step (2.4.1)'', meaning that the conflict cannot be resolved, and the parser must still be rejected by Test ART2.

Chapter 6

Results and Conclusions

6.1. Introduction

In order to test the viability of the parser generation methods proposed in this thesis, an SE-SAR-NSLR(1)† parse-table generator was written in Modula-2. The program is based on an SLR(1) parse-table generator written by Gordon Cormack. A user's guide for this program is provided in Appendix F.

The language description concepts of this thesis were tested by preparing an SE-SAR-NSLR(1) grammar for ISO Pascal. The grammar is adapted from the one given by Jensen and Wirth[26] and is presented in Appendix D. An SE-SAR-NSLR(1) grammar for Modula-2 has also been written, as well as grammars for small languages such as the input language for the parser generator. The recognizers generated from the Pascal and Modula-2 grammars were verified on a test suite of programs that was originally written to test student compilers for a course on compiler construction (CS-444). The Modula-2 recognizer was also verified on the source code for the parser generator itself.

The remainder of this chapter discusses experience gained from the implementation of the proposed techniques, and its apparent advantages and disadvantages.

6.2. Characteristics of the Sample SE-SAR-NSLR(1) Grammar

An examination of the single-phase grammar for Pascal presented in Appendix D, henceforth called *Grammar D*, illuminates its salient characteristics. As is to be expected, most of the nontrivial rules in the grammar come directly from the Jensen-Wirth grammar. Grammar *D* uses only three compound adjacency-restriction rules, which describe 1334 simple adjacency restrictions, and occupy 10 lines of the grammar. The grammar also uses 35 exclusion rules, occupying 4 lines. The purpose of all but one of these disambiguation rules is quite clear and can be seen immediately to address one of the ambiguities discussed in Chapter 1:

1) the identifier-keyword ambiguity,
2) the dangling-else ambiguity, or
3) the longest-match ambiguity between:
   a) identifiers and keywords
   b) identifiers and numeric constants, and
   c) keywords and numeric constants.

The only slightly obscure rule is

$$\textit{StarNoRRound} \not\leftarrow \text{')'}.$$

The purpose of this rule is to recognize correctly the compound comment-closure symbol "`*)`". Asterisks appearing as part of a comment rather than part of the closing delimiter must be reduced through the symbol *StarNoRRound*. The given adjacency restriction prevents this reduction if the asterisk is immediately followed by a right parenthesis.

The grammar presented has a total of 560 rules. This is about 1.9 times the number of rules of the LEX-YACC grammars presented in Appendix A. A quick comparison of the single-phase grammar with the two-phase LEX-YACC grammars shows the reason for the greatest part of this difference. In the LEX grammar, the description of the symbol `letter` is given in one rule

† Recall that this acronym stands for *simple exclusion, simple adjacency restriction, noncanonical simple LR(1).*

```
letter      [a-zA-Z]
```

whereas in Grammar *D* this requires 52 rules. Similar differences appear in the description of the symbols *Digit*, *StringElement*, and *CommentElement*. In particular a valid string character can be described by the LEX pattern "[^'\n]", meaning anything that is not an apostrophe or a newline character, whereas in Grammar *D* each valid character of a string must be given by a separate rule. Note, however, that the LEX pattern must be implemented with about the same number of state transitions as for the rules of Grammar *D*, and the difference lies only in level of metasyntactic sugar provided by LEX. Such abbreviations could be easily built into the proposed parser generator, but have not yet been provided. Another cause for the bulk of the unified grammar, its treatment of white space, is discussed in detail in the next section of this chapter.

One source of bulkiness in Grammar *D* that is not superficial is the treatment of the reserved keywords. Each keyword in Pascal has a corresponding symbol in Grammar *D*, in order that that symbol can appear in the adjacency restrictions for solving the longest-match ambiguity with other keywords and with identifiers. This source of bulkiness seems to be permanent since it is difficult to conceive of a way of disambiguating language constructs without giving names to those constructs.

Once the grammars of Appendices A and D are processed they yield parsing automata of very similar size. The LEX and YACC automata have a combined total of 914 states, whereas the NSLR(1) parsing automaton has 1005 states. These close numbers are further evidence that the differences in the sizes of the two grammars are primarily due to superficial syntactic differences.

The most important thing to note about Grammar *D* is that it completely and unambiguously describes the syntax of Pascal (those aspects of the syntax that are context-free), it is machine processable, and yet it is not unreasonably long.

6.3. The Treatment of White Space

In Grammar *D*, the treatment of white space is modeled after the treatment by two-phase parsers, and those accustomed to writing traditional two-phase parsers might consider this to be the clearest approach. The symbol *White* describes what may constitute optional white space in Pascal—any sequence of blanks, tabs, newlines, formfeeds, and comments, or nothing at all. The symbols of Grammar *D* whose names are entirely in uppercase, such as *BEGIN* and *ASSIGN*, each generate strings that would be recognized as tokens by a two-phase recognizer. All of the token symbols have similar descriptions: they generate a pretoken symbol that does not include white space, followed by the symbol *White*.

Another possible treatment of white space would be to distribute the symbol for optional white space throughout the grammar, between the symbols that it may separate. For example, instead of the rule:

CompountStatement = BEGIN StatementSequence END.

one could write the rule:

CompoundStatement = BEGIN White StatementSequence White END.

Such a scheme would reduce the number of rules and symbols in the grammar, at the cost of making many of the rules longer. In experiments with this approach, it was found the the number of parser states generated by a grammar using this approach was higher than for Grammar *D*. Since the number of symbols decreased, and the number of states increased, the total parse-table size remained about the same.

To make the approach of using distributed white space a reasonable one, a very short and nonobtrusive name should be given to the symbol representing optional white space. A name consisting of a single underscore character seems to be a suitable choice, and so the rule above would take the form:

CompoundStatement = BEGIN _ StatementSequence _ END.

The sample rule given above illustrates a difficult problem with distributing white space throughout the grammar. In Pascal, a *StatementSequence* can be empty, and so in the the sample rule we have three symbols in a row that can generate ϵ. The grammar is therefore ambiguous, and it is a kind of ambiguity that is difficult to correct in an SE-SAR-NSLR(1) grammar. As a result of these disadvantages, the white-space treatment of Grammar *D* was the one chosen for presentation in this thesis.

Some readers may consider both of the above approaches to describing white space as too cumbersome, and so let us discuss a third possibility. The symbols in a grammar for a programming language can be divided into two classes: those that can contain white space and those that cannot. In addition, those symbols that can contain white space, usually do so in a quite regular way: white space is usually allowed anywhere between immediate descendants. It might therefore be possible to develop a metasyntax which simply lists the two classes of symbols, and a parser generator that takes care of distributing white space as needed. The user specified productions would be automatically modified to include the white space as requested, and parser generation would then proceed as usual. This third approach leads to parsers very similar to the second approach above, and hence also awaits more powerful parser generation techniques.

6.4. Single-phase Versus Two-Phase Parsing

Chapter 3 presented a comparison of two two-phase recognizers: PRLY, an FSM-DPDA recognizer, and PRYY, a DPDA-DPDA recognizer. The conclusion of that chapter was that the descriptions and implementations were of similar sizes, and running times.

In this section, we compare two-phase recognizers with a single-phase recognizer prepared according to the methods of Chapters 4 and 5. Grammar *D* and the parser it generates can be compared directly to the grammars and parsers of appendices A through C, but some aspects of the comparison are meaningless. The rule counts can be compared, but since Grammar *D* uses a different philosophy from the other grammars (one of explicitly resolving all ambiguities), and since LEX and YACC have evolved some short-hand forms for common rules, this comparison will be misleading. The number of parser states could also be compared, but since this may depend on the amount of table optimization and compression performed, this comparison too may be faulty. Finally the running times for the parser generator and the generated parsers are incommensurate, since these are sensitive to the source and target language used and to implementation strategies.

To make a comparison between two-phase and single-phase recognizers that is as meaningful as possible, a third two-phase recognizer has been prepared that uses the same input description language and implementation details as the single-phase recognizer of this thesis. The grammar for this third two-phase recognizer is given in Appendix E. The method of comparison will be to adjust the results obtained by processing the grammars of appendix E so that they are the same as those obtained from the YACC-YACC generated recognizer, PRYY, and use the same factors to normalize the results obtained from Grammar *D*. In this way we hope to normalize away differences in implementation strategies and source and target languages of PRYY and the recognizer generated by Grammar *D*.

The basis of comparison is thus as follows. The recognizers PRLY and PRYY can be compared because they are both mature products with similar implementation strategies. The recognizer prepared from the two-phase grammars of Appendix E will be evaluated and the ratio of its performance statistics as compared to PRYY will be used to normalize the performance of a recognizer prepared from Grammar *D*. This is equivalent to multiplying the performance statistics of PRYY by the entries in the ratio column of Table 6.1 to get the expected performance of a single phase recognizer written using the programming techniques of YACC. The normalized performance statistics of the single-phase recognizer generated by Grammar *D* will then be compared to the performance statistics of the standard recognizer model, PRLY.

There are two SE-SAR-NSLR(1) grammars in Appendix E, Grammar E_1 and Grammar E_2, derived by splitting Grammar D into two parts. Grammar E_1 was intended to be as similar as possible to the scanner grammar for PRYY given in Appendix B. Grammar E_2 was intended to be as similar as possible to the standard parser grammar used for PRLY and PRYY given in Appendix A.

The language recognized by Grammar E_1, is a stream of Pascal tokens optionally separated by white space. All of the disambiguation rules of Grammar D, except the one for resolving the dangling-else ambiguity, are also needed in Grammar E_1: the longest-match ambiguities and the reserved-keyword ambiguity must still be resolved. In addition, since Grammar E_1 has no information about which tokens may legally follow which other tokens (all such information being contained in Grammar E_2) many additional disambiguation rules must be added. For instance we must add the rule *COLON* $\not\dashv$ *EQUAL* so that the first phase of the recognizer will know that the sequence ":=" should be parsed as *ASSIGN* and not as *COLON* and *EQUAL*. Thus Grammar E_1 requires six new compound adjacency restrictions, representing 33 simple adjacency restrictions.

Table 6.1 summarizes the comparison of a two-phase SE-SAR-NSLR(1) Pascal recognizer, with a single-phase SE-SAR-NSLR(1) recognizer.

	Two-Phase Recognizer			One-Phase Recognizer	Ratio
	Scanner	Parser	Total		
Symbols	207	169	376	343	0.91
Rules	380	210	590	560	0.95
States	450	358	808	1005	1.24
Time to generate tables (seconds)†	29.5	9.3	38.8	119.8	3.09
Total parser actions	93,150	60,502	153,652	344,715	2.24
Total non-**error** parser actions	40,076	2,786	42,862	59,389	1.39
Parse time‡			58.9	49.9	0.85

† The tests were run on a VAX 8650.

‡ Time to parse 8,564 lines of Pascal code run on a Micro-VAX II.

Table 6.1: One-Phase Versus Two-Phase Syntax Analysis

From this table we can see that the two-phase recognizer has a slight advantage in the number of rules and symbols. Thus the grammar writer not only benefits in not having to partition his grammar, but also in being able to work with a smaller total grammar. The rest of the statistics are not as favourable. The two-phase recognizer suffers from a small increase in the number of parser states and non-**error** parser actions. More significantly, the two-phase recognizer suffers from a very large increase in the time required to generate the parser, and in the total number of parser actions.

The reason for the increase in the parser generation time is that it is roughly proportional to $r \cdot t \cdot y^3$ where: r = number of rules, t = number of states, and y = number of symbols. Since a product of sums is larger than a sum of products, the single-phase parser takes longer to generate even though r and y are smaller. There is some evidence to believe that a change in the data structures of the parser generator could give a parser generation time that is proportional to $r \cdot t \cdot y^2$, so that even though the single-phase recognizer will still take longer to generate, the ratio of the comparison to the two-phase recognizer will improve.

The Speed of One-Phase Parsers

Table 6.1 shows an improvement in the running time of the single-phase parser over the two-phase parser. There are two principal reasons for this improvement. The first is that the two-phase parser performs some unnecessary work in reducing the input to a token stream, whereas the single-phase parser reduces low-level symbols to high-level ones without an intermediate conversion to a token stream. The second reason is that the procedure-call interface between the two phases is eliminated in the single-phase parser.

The Standard Recognizer Versus a One-Phase Recognizer

Table 6.2 presents a comparison of the three recognizer techniques after applying the normalization, described above, of the results for the one-phase recognizer. Specifically, the size and time entries of Table 6.1 were multiplied by the ratio of the entries for the YACC-YACC recognizer of Table 3.1 to the entries for the two-phase recognizer of Table 6.1. All values are rounded to two significant digits since the method of comparison is probably no more precise than that.

	Rules	States	Object Size (Bytes)	Parse Time* (Sec.)
LEX-YACC	1.0	1.0	1.0	1.0
YACC-YACC	2.0	0.9	0.7	1.4
One-Phase	1.9	1.1	1.0	1.2

Table 6.2: Ratios of size and speed of the YACC-YACC Pascal recognizer, PRYY, and a one-phase SE-SAR-NSLR(1) Pascal recognizer to the standard LEX-YACC recognizer, PRLY.

The table shows that the one-phase techniques proposed are competitive with the standard recognizer. There are compiler writing tools that generate faster recognizers than LEX and YACC, but the comparison with LEX and YACC is still a valid one. Our recognizer uses straightforward implementation techniques, so it is valid to compare it with other tools that also use straightforward techniques as do LEX and YACC. When an optimizing parser generator is developed for SE-SAR-NSLR(1) grammars, then its results can be compared with other optimizing parser and scanner generators such as FLEX[45] and BISON.[17] An interesting property of the normalization process is that it will yield the same table independent of such processing details as whether or not the optimizer option is selected when compiling the one-phase recognizer.

Why has the single-phase recognizer produced results that are so similar to the standard recognizer, PRLY? Conventional wisdom would tell us that a pushdown automaton should be slower than a finite-state automaton. A closer analysis yields the following points:

- Theory tells us that an FSM, a DPDA, and a noncanonical DPDA[55] all run in linear time on the length of the input, so the performance statistics can only differ by a constant factor.
- The stack operations push and pop, the main difference between an FSM and a pushdown automaton, are not expensive. Each can be implemented as a single data move, and a pointer increment. In comparison, a table access for an FSM or PDA state transition may require a multiplication, to convert two-dimensional indices into a one-dimensional index, or may require a table search, if the table is stored as a sparse matrix.
- The interface between the FSM scanner and the DPDA parser in the standard model, is probably implemented as a procedure call. On modern computers a procedure call commonly involves the pushing and popping of a large amount of information such as registers, display, and local storage.

- The FSM scanner of the standard recognizer may require $k > 1$ lookahead characters. For Pascal for instance $k = 2$. Multiple lookahead complicates table accesses further.

6.5. The Independence of RCFG's and One-Phase Parsing

There are two principal topics of this thesis: the use of restricted context-free grammars to describe programming languages, and the implementation of single-phase parsers for programming languages. It should be pointed out that these two methods are not necessarily bound together. It is possible to use RCFG's to describe programming languages, and thus reap the benefits of a complete, precise, and unambiguous description, and yet still implement the described language by the traditional two-phase method.

It is also possible to use the traditional imprecise description of programming languages in publications, but use SE-SAR-NSLR(1) grammars and parsers for the actual implementation.

6.6. Future Work

There have been hints throughout this thesis of how the proposed metalanguage and parser generation scheme could be improved. This section summarizes possible improvements, and suggests directions for new research.

6.6.1. Noncanonical Nonsimple LR Parsers

The most significant improvement to the power of the parser generator would be to devise a noncanonical LALR (NLALR) parser generator to replace the NSLR strategy described by this thesis. As with a canonical LALR parser, in an NLALR parser the lookahead symbol for each complete item is not simply the FOLLOW set of the left part of the item, but rather depends on the paths through the parsing automaton to the current state. If the symbol Z is in the lookahead set of a complete item $[A \rightarrow \alpha \bullet]$ in a state s, that means that some state t on a path to s must contain an item $[B \rightarrow \bullet \beta\gamma]$, such that A is in LAST(β) and Z is in FIRST(γ). In addition, in an NLALR parser, no two states have the same core item set (the items of a state excluding the lookahead information).

There are two principal difficulties in implementing an NLALR parser. The first relates to determining lookahead sets for items added during state expansion. State expansion is performed when there is a conflict on a lookahead symbol, and the effect of state expansion is to shift on the conflicting lookahead symbol. If the conflicting symbol is A, then for all rules of the form $B \rightarrow \beta$, such that A is in FIRST(β), an item $[B \rightarrow \bullet \beta]$ is added to the state. The customary method of handling LALR items, as presented by Aho, Sethi, and Ullman[5] for instance, is to attach a lookahead set to each item, and add symbols to the set as the parser is being built, and thus new items added by state expansion too will need lookahead sets attached to them.

The essence of the problem is that we cannot know in advance which symbols will be removed from lookahead sets by state expansion, and shifted by **shift** actions. As a result, we would need to collect the LA follow set (as opposed to the simple follow set) of all symbols in all lookahead sets as we were building the parser. Furthermore, any symbol in a lookahead set of an item resulting from state expansion, could itself be removed from a lookahead set by a later state expansion, and its LA lookahead set also needed. We would therefore need to compute LALR(k) lookahead sets for some unknown k, even though only one symbol of lookahead would actually be used during parsing.

Compound Nonsimple-Simple LR Parsers

A simple solution to this problem is to use the LA lookahead set for ordinary items, but the simple lookahead set for items added during state expansion. Kusters[34] has implemented this method. The method works, but weakens the power of the grammars somewhat. A better solution would be to collect lookahead sets for regular items only, and when an expansion item is added, retraverse the parsing automaton to collect its lookahead set. The problem lies in finding efficient ways to perform this traversal. The work of DeRemer and Pennello;[15] and of Park, Choe and Chang[43] may be useful in formulating other ways of collecting the lookahead information needed.

On the Termination of the Construction for Nonsimple Noncanonical LR Parser Generators

The second difficulty in constructing an NLALR parser arises in testing the consistency of states. In an LALR parser, all states and paths between states must be constructed before the consistency of a state can be tested. But in an NLALR parser generator, an inconsistent state may lead to state expansion, and state expansion may generate a transition to an existing state s. The new transitions to state s may cause the lookahead sets of some items to grow. Adding new lookahead symbols would require that the consistency of state s, and all states on a path from s, be checked again. Each consistency check may require another state expansion. Thus state expansion is not a once-per-state operation as it is for an NSLR parser.

Despite the fact that states may have to be visited repeatedly for state expansion, it is possible to write parser generators that are guaranteed to terminate. This conclusion can be seen by noticing that in an NLALR parser, the states have unique core item sets. Since the number of rules in the grammar is finite, and the number of items that can be generated from each rule is finite, the total number of items that can be generated is finite. Since no item may be repeated in a state, the number of unique states is finite. Also note that since the number of symbols in a grammar is finite, the size of any lookahead set must also be finite. The finiteness of the parser to be generated is a good indicator that an algorithm to compute such a parser should terminate.

The one way that state expansion could loop forever is if the same symbol deleted from a lookahead set by a state expansion was again added because of a new path added to the state by the expansion. The number of states involved in such a cycle must be finite, so such a processing cycle could be detected, and the grammar rejected. Alternatively cycles could be prevented by ensuring that consistency of a state is only checked when a new path to a state is created not when one is deleted. Since the number of paths possible to a state is finite, state expansion would not go on forever.

Can a noncanonical full LR(1) parser also be generated? Preliminary analysis indicates that the answer is yes. The time and space required for parser generation will be greater, however, as will the space for the generated parser. Spector[52] argues, that with table compression, the time and space for LR(1) parser generation need not be significantly greater than for an LALR(1) parser, but that the benefits are significant for the grammar writer. He and Pager[42] present practical methods for generating LR(1) parsers. It may be possible to adapt these methods to noncanonical LR(1) parser generation.

It can be shown, in a similar fashion to the proof for NLALR parser generation, that NLR parser generation can be made to terminate. In this case, the states do not have unique core sets, but each state is unique when both the core items and the lookahead sets are considered. Since the number of productions and symbols is finite, the number of unique states that can be generated, and the paths between them, are finite, and hence parser generation can eventually terminate.

6.6.2. Nonsimple Restrictive Rules

It is also possible to extend the treatment of exclusion rules and adjacency restriction rules to relax the constraints on their usage. Consider for instance the treatment of the exclusion rule $A \nrightarrow B$. The parser generator described in this thesis decides whether a complete item is concluding the recognition of the symbol A without examining the paths to the state containing the item. A parser-generation algorithm that analyzed the paths to each state to determine the true predecessors of each item could be more accurate and accept more grammars. The same is true for the treatment of adjacency restrictions. A possible name for these larger classes of grammars would be LBE (look-back exclusion) and LBAR (look-back adjacency restriction) grammars.

6.6.3. More Powerful Disambiguation Rules

The disambiguation rules proposed in this thesis are not the only ones possible. It is possible that a single disambiguation rule could replace both of these.

There are also some ambiguity problems that the rules proposed here do not handle well. For instance, Aho and Johnson present a disambiguation rule for concisely specifying operator precedence and transitivity. The disambiguation rules proposed in this thesis are poor at handling this problem, and produce a grammar almost as large as disambiguation by grammar rewriting.

6.6.4. Parse Table Compression

Single-phase grammars describe programming languages right down to the character level. In a programming language there are groups of characters such as letters and digits, that can be treated identically by a parser. This repetition should allow for parser compression techniques that would be of minimal benefit in other types of parsers whose terminal symbols are significantly more diverse.

6.6.5. A Reliable Compiler Writing Tool

This thesis has proposed several novel techniques for describing and implementing programming language translators. There is bound to be a great deal of resistance to these ideas until they are implemented in a complete and reliable compiler writing tool intended for widespread use. The implementation of an SE-SAR-NSLR(1) parse-table generator reported above is a step toward this goal, and work on a complete parser generator to rival LEX and YACC is in progress.

Appendix A

The Standard Pascal Recognizer

This appendix contains the description of a Pascal recognizer, prepared according to standard methods, consisting of a finite state scanner and a DPDA parser. The description consists of the grammar for the standard scanner, the grammar for the standard parser, the C-language description of the interface between the scanner and the parser, and the makefile showing the compilation dependencies of the two modules.

A.1. The Grammar for the Standard Scanner

```
%{
/*
 *  s c a n n e r . l
 *  =================
 *
 *  Lex script for a Pascal token scanner.
 */

# include "tokens.h"
# include "extern.h"

%}

%Start  CODE  COMMENT

letter          [a-zA-Z_]
digit           [0-9]
a               [aA]
b               [bB]
c               [cC]
d               [dD]
e               [eE]
f               [fF]
g               [gG]
h               [hH]
i               [iI]
j               [jJ]
k               [kK]
l               [lL]
m               [mM]
n               [nN]
o               [oO]
p               [pP]
q               [qQ]
r               [rR]
s               [sS]
t               [tT]
u               [uU]
```

```
v                [vV]
w                [wW]
x                [xX]
y                [yY]
z                [zZ]

%%

                         /* Enter scanner in CODE state. */
                         BEGIN CODE;

<CODE>"{"                BEGIN COMMENT;
<COMMENT>"}"             BEGIN CODE;
<COMMENT>.               ;
<COMMENT>\n              ;
<CODE>\n                 ;
<CODE>[ \t\f]            ;       /* Skip white space. */
<CODE>[-+*/=><:;,.()\[\]^] {     /* Single character tokens
                                  * represent themselves. */
                            return((int) *yytext);}
<CODE>"(."               return((int) ']');
<CODE>".)"               return((int) '[');
<CODE>{a}{n}{d}          return(AND);
<CODE>{a}{r}{r}{a}{y}    return(ARRAY);
<CODE>":="               return(ASSIGN);
<CODE>{b}{e}{g}{i}{n}    return(BEGIN_);
<CODE>{c}{a}{s}{e}       return(CASE);
<CODE>{c}{o}{n}{s}{t}    return(CONST);
<CODE>{d}{o}             return(DO);
<CODE>".."               return(DOTDOT);
<CODE>{d}{o}{w}{n}{t}{o}        return(DOWNTO);
<CODE>{e}{l}{s}{e}       return(ELSE);
<CODE>{e}{n}{d}          return(END);
<CODE>{f}{o}{r}          return(FOR);
<CODE>{f}{u}{n}{c}{t}{i}{o}{n}  return(FUNCTION);
<CODE>">="               return(GE);
<CODE>{g}{o}{t}{o}       return(GOTO);
<CODE>{d}{i}{v}          return(IDIV);
<CODE>{i}{f}             return(IF);
<CODE>{i}{n}             return(IN);
<CODE>{digit}+           return(INT);
<CODE>"<="               return(LE);
<CODE>{l}{a}{b}{e}{l}    return(LABEL);
<CODE>{m}{o}{d}          return(MOD);
<CODE>"<>"               return(NE);
<CODE>{n}{i}{l}          return(NIL);
<CODE>{n}{o}{t}          return(NOT);
<CODE>{o}{f}             return(OF);
<CODE>{o}{r}             return(OR);
<CODE>"@"                return((int) '^');
<CODE>{p}{a}{c}{k}{e}{d}        return(PACKED);
<CODE>{f}{i}{l}{e}       return(PHYLE);
<CODE>{p}{r}{o}{c}{e}{d}{u}{r}{e} { return(PROCEDURE);}
<CODE>{p}{r}{o}{g}{r}{a}{m} {
```

```
                          return(PROGRAM);}
<CODE>{digit}+"."{digit}+({e}[-+]?{digit}+)?     |
<CODE>{digit}+{e}[-+]?{digit}+  return(REAL);
<CODE>{r}{e}{c}{o}{r}{d}        return(RECORD);
<CODE>{s}{e}{t}          return(SET);
<CODE>'([^'\n]|'')+'     return (STRING);
<CODE>{r}{e}{p}{e}{a}{t}        return(REPEAT);
<CODE>{t}{h}{e}{n}       return(THEN);
<CODE>{t}{o}             return(TO);
<CODE>{t}{y}{p}{e}       return(TYPE);
<CODE>{u}{n}{t}{i}{l}    return(UNTIL);
<CODE>{v}{a}{r}          return(VAR);
<CODE>{w}{h}{i}{l}{e}    return(WHILE);
<CODE>{w}{i}{t}{h}       return(WITH);

<CODE>{letter}({letter}|{digit})*       return(ID);
<CODE>.                  return(GARBAGE);
%%
```

A.2. The Grammar for the Standard Parser

```
%token <text> ID
%token <int_const> INT
%token <real_const> REAL
%token <text> STRING
%token PROGRAM BEGIN_ END
%token LABEL CONST TYPE VAR PROCEDURE FUNCTION
%token FILE_ PACKED ARRAY RECORD SET OF
%token ASSIGN DOTDOT NE LE GE IN IDIV MOD
%token AND OR NOT NIL
%token IF THEN ELSE WHILE DO FOR TO DOWNTO
%token CASE REPEAT UNTIL WITH GOTO GARBAGE
%union {
       double float_const;
       int    int_const;
       char   *text;
       }
%%

program          : program_head block '.' ;

program_head     : PROGRAM ID program_parms ';' decls ;

program_parms    :
                 | '(' file_list ')' ;

file_list        : ID
                 | file_list ',' ID ;

decls            : label_decl_part const_decl_part
                   type_decl_part var_decl_part
                   proc_decl_part ;
```

```
block           : prologue_part compound_stat
                  epilogue_part ;

prologue_part   : ;

epilogue_part   : ;

label_decl_part :
                | LABEL label_decl_list ';' ;

label_decl_list : label_decl
                | label_decl_list ',' label_decl ;

label_decl      : INT ;

const_decl_part :
                | CONST const_decl_list ';' ;

const_decl_list : const_decl
                | const_decl_list ';' const_decl ;

const_decl      : ID '=' const ;

const           : unsigned_num
                | '+' unsigned_num | '-' unsigned_num
                | ID | '+' ID | '-' ID
                | STRING ;

unsigned_num    : INT | REAL ;

type_decl_part  :
                | TYPE type_decl_list ';' ;

type_decl_list  : type_decl
                | type_decl_list ';' type_decl ;

type_decl       : ID '=' type ;

type            : simple_type | structured_type
                | '^' ID ;

simple_type     : scalar_type | ID ;

scalar_type     : '('  scalar_list  ')'
                | const DOTDOT const ;

scalar_list     : ID | scalar_list ',' ID ;

structured_type : u_struct_type | PACKED u_struct_type ;

u_struct_type   : ARRAY '['   array_rest
                | RECORD field_list END
                | SET OF simple_type
                | FILE_ OF type ;
```

```
array_rest      : simple_type ']' OF type
                | simple_type ',' array_rest ;


field_list      : fixed_part | fixed_part ';'
                | fixed_part ';' variant_part
                | variant_part | ;

fixed_part      : fixed_item_list
                | fixed_part ';' fixed_item_list ;

fixed_item_list : ID ':' type
                | ID ',' fixed_item_list ;

variant_part    : CASE tag_field OF variant_list
                | CASE tag_field OF variant_list ';' ;

tag_field       : ID | ID ':' ID ;

variant_list    : variant | variant_list ';' variant ;

variant         : case_label_list ':' '(' field_list ')' ;

case_label_list : const | case_label_list ',' const ;

var_decl_part   :
                | VAR var_decl_list ';' ;

var_decl_list   : var_decl
                | var_decl_list ';' var_decl ;

var_decl        : ID ':' type
                | ID ',' var_decl ;

proc_decl_part  :
                | proc_decl_list ;

proc_decl_list  : proc_decl
                | proc_decl_list proc_decl ;

proc_decl       : proc_heading block ';'
                | proc_beg f_parm_decl ';' ID ';'
                | func_beg f_parm_decl ':' ID ';' ID ';' ;

proc_heading    : proc_head_beg decls
                | func_head_beg decls
                | func_beg ';' decls ;

proc_head_beg   : proc_beg f_parm_decl ';' ;

func_head_beg   : func_beg f_parm_decl ':' ID ';' ;

proc_beg        : PROCEDURE ID ;
```

```
func_beg        : FUNCTION ID ;

f_parm_decl     : '(' f_parm_list ')' | ;

f_parm_list     : f_parm
                | f_parm_list ';' f_parm ;

f_parm          : val_fparm_list
                | VAR var_fparm_list
                | func_beg f_parm_decl ':' ID
                | proc_beg f_parm_decl ;

val_fparm_list  : ID ':' type
                | ID ',' val_fparm_list ;

var_fparm_list  : ID ':' type
                | ID ',' var_fparm_list ;

compound_stat   : BEGIN_ stat_list END ;

stat_list       : stat | stat_list ';' stat ;

stat            : ul_stat | label ul_stat ;

label           : INT ':' ;

ul_stat         : simple_stat | struct_stat | ;

simple_stat     : beg_stat var ASSIGN expr
                | beg_stat proc_invok
                | no_hassel_stat
                | GOTO INT ;

beg_stat        : ;

proc_invok      : noparms_pinvok | plist_pinvok ')'
                | noparms_pinvok '(' ')' ;

noparms_pinvok  : ID ;

plist_pinvok    : noparms_pinvok '(' parm
                | plist_pinvok ',' parm ;

var             : ID | var '.' ID
                | subscripted_var ']' | var '^' ;

subscripted_var : var '[' expr
                | subscripted_var ',' expr ;

parm            : expr | expr ':' expr
                | expr ':' expr ':' expr ;

expr            : simple_expr
```

```
                | expr '=' simple_expr
                | expr NE simple_expr
                | expr LE simple_expr
                | expr '<' simple_expr
                | expr GE simple_expr
                | expr '>' simple_expr
                | expr IN simple_expr ;

simple_expr     : term | '+' term | '-' term
                | simple_expr '+' term
                | simple_expr '-' term
                | simple_expr OR term ;

term            : factor
                | term '*' factor | term '/' factor
                | term IDIV factor | term MOD factor
                | term AND factor ;

factor          : var | unsigned_const | '(' expr ')'
                | func_invok | set | NOT factor ;

unsigned_const  : unsigned_num | STRING | NIL ;

func_invok      : plist_finvok  ')'
                | start_finvok '(' ')' ;

plist_finvok    : start_finvok '(' parm
                | plist_finvok ',' parm ;

start_finvok    : ID ;

set             : '[' element_list ']' | '[' ']' ;

element_list    : element | element_list ',' element ;

element         : expr | expr DOTDOT expr ;

struct_stat     : if_then_else stat
                | if_beg stat
                | while_beg DO stat
                | for_beg DO stat
                | with_beg DO stat ;

if_then_else    : if_beg matched_stat ELSE ;

if_beg          : IF expr THEN ;

while_beg       : WHILE expr ;

for_beg         : for_init updown expr ;

for_init        : FOR ID ASSIGN expr ;

updown          : TO | DOWNTO ;
```

```
with_beg        : WITH rec_list ;

no_hassel_stat  : compound_stat
                | case_alt stat ';' END
                | case_alt stat END
                | repeat_beg stat_list UNTIL expr ;

repeat_beg      : REPEAT ;

case_beg        : CASE expr OF const
                | case_beg ',' const
                | case_alt stat ';' const ;

case_alt        : case_beg ':' ;

matched_stat    : ul_m_stat | label ul_m_stat ;

ul_m_stat       : simple_stat
                |
                | if_then_else matched_stat
                | while_beg DO matched_stat
                | for_beg DO matched_stat
                | with_beg DO matched_stat ;

rec_list        : var | rec_list ',' var ;
```

A.3. The Interface between the Scanner and the Parser

The two files "tokens.h" and "extern.h" describe the interface between the standard scanner and the standard parser.

A.3.1. The Automatically-Produced file "tokens.h"

```
typedef union  {
       double float_const;
       int    int_const;
       char   *text;
       } YYSTYPE;
extern YYSTYPE yylval;
# define ID 257
# define INT 258
# define REAL 259
# define STRING 260
# define PROGRAM 261
# define BEGIN_ 262
# define END 263
# define LABEL 264
# define CONST 265
# define TYPE 266
# define VAR 267
# define PROCEDURE 268
# define FUNCTION 269
# define FILE_ 270
```

```
# define PACKED 271
# define ARRAY 272
# define RECORD 273
# define SET 274
# define OF 275
# define ASSIGN 276
# define DOTDOT 277
# define NE 278
# define LE 279
# define GE 280
# define IN 281
# define IDIV 282
# define MOD 283
# define AND 284
# define OR 285
# define NOT 286
# define NIL 287
# define IF 288
# define THEN 289
# define ELSE 290
# define WHILE 291
# define DO 292
# define FOR 293
# define TO 294
# define DOWNTO 295
# define CASE 296
# define REPEAT 297
# define UNTIL 298
# define WITH 299
# define GOTO 300
# define GARBAGE 301
```

A.3.2. The File "extern.h"

```
/*
 *  extern.h
 *  ========
 *
 *  External variable declarations.
 */

extern int yylineno;     /* Current source line no. */
extern char yytext[];    /* Current token string. */
extern YYSTYPE yylval;   /* Current token value. */
```

A.4. The Makefile for Synchronizing Generation of the Recognizer

```
#
#  Makefile
#  ========
#
#
#  Possible C flags:
#  -O           - C optimizer.
```

```
#  -g           - Include globals in object for xdb.
#  -pg          - Extensive profiling.
CFLAGS = -O

prly :   main.o parser.o scanner.o yyerror.o
        cc $(CFLAGS) main.o parser.o scanner.o yyerror.o -ll \
             -o prly

scanner.c : scanner.l tokens.h extern.h
        lex -t scanner.l >scanner.c

# Tokens.h is generated by the parser generator.
tokens.h :
        make parser.c

parser.c :  parser.y extern.h
        yacc -d -v parser.y
#
#       -- Replace tokens.h only if it has changed.
#       -- This prevents useless recompiles of scanner.c.
        @csh -f diff_repl y.tab.h tokens.h
        @mv y.output parser.graph
        @mv y.tab.c parser.c
#
#       -- Replace tokens.h only if it has changed.
#       -- This prevents useless recompiles of scanner.c.
        @csh -f diff_repl y.tab.h tokens.h
```

Appendix B

A Single-Metalanguage Two-Phase Pascal Recognizer

This appendix contains the description of a single-metalanguage two-phase Pascal recognizer. Only the BNF grammar for the scanner is presented, since the grammar for the parser and the description of the scanner-parser interface are the same as for the standard recognizer presented in Appendix A. The grammar is processed by YACC to produce a scanner in the C language.

Due to the difficulty of calling a YACC generated scanner as a procedure, the scanner and the parser run as two separate processes and communicate tokens over a UNIX pipe. The details of this method are not shown here as they are not relevant to this thesis. The assumption is that if this technique of describing the scanner with the same metalanguage as is used to describe the parser became popular, then a version of YACC that produced callable scanners would be developed.

B.1. The Grammar for the Scanner

```
/*
 *  YaccScn.y
 *  =========
 *
 *  Yacc script for a Pascal token scanner.
 *
 *  Difference from lex version:
 *  Elipsis in a range cannot immediately follow a digit.
 */

%{

# include <stdio.h>
# include "tokens.h"
# include "pryy_macros.h"

# define SET_TOK(arg)    token_code = arg

# define SEND_TOKEN      *wrptr++ = token_code;\
                         token_code = 0

extern short int *wrptr;

int token_code;
int chin;
extern int lineno;
%}

%start good_scan

%%
good_scan       : token_stream
                  { *wrptr++ = 0; /* End of tokens. */ }
                | token_stream  act_white
                  { *wrptr++ = 0; /* End of tokens. */ } ;
```

```
act_white       : ' ' | '\t' | '\f' | '\n' | comment
                | act_white ' ' | act_white '\t'
                | act_white '\f' | act_white '\n'
                | act_white comment ;

token_stream    : /* nothing */
                | token_stream  token { SEND_TOKEN; }
                | an_token_stream  token { SEND_TOKEN; }
                | num_token_stream  token { SEND_TOKEN; }
                | token_stream  act_white  token
                  { SEND_TOKEN; }
                | an_token_stream  act_white  token
                  { SEND_TOKEN; }
                | num_token_stream  act_white  token
                  { SEND_TOKEN; } ;

an_token_stream : token_stream  an_token { SEND_TOKEN; }
                | token_stream  act_white  an_token
                  { SEND_TOKEN; }
                | an_token_stream  act_white  an_token
                  { SEND_TOKEN; }
                | num_token_stream  an_token
                  { SEND_TOKEN; }
                | num_token_stream  act_white an_token
                  { SEND_TOKEN; } ;

num_token_stream
                : token_stream  num_token
                  { SEND_TOKEN; }
                | token_stream  act_white  num_token
                  { SEND_TOKEN; }
                | an_token_stream  act_white  num_token
                  { SEND_TOKEN; }
                | num_token_stream  num_token
                  { SEND_TOKEN; }
                | num_token_stream  act_white num_token
                  { SEND_TOKEN; } ;

token           : STRING_
                | '-'     { SET_TOK('-'); }
                | '+'     { SET_TOK('+'); }
                | '*'     { SET_TOK('*'); }
                | '/'     { SET_TOK('/'); }
                | '='     { SET_TOK('='); }
                | '<'     { SET_TOK('<'); }
                | '>'     { SET_TOK('>'); }
                | ':'     { SET_TOK(':'); }
                | ';'     { SET_TOK(';'); }
                | ','     { SET_TOK(','); }
                | '['     { SET_TOK('['); }
                | '(' '.' { SET_TOK('['); }
                | ']'     { SET_TOK(']'); }
                | '.' ')' { SET_TOK(']'); }
                | '^'     { SET_TOK('^'); }
```

```
                | '@'     { SET_TOK('^'); }
                | '('     { SET_TOK('('); }
                | ')'     { SET_TOK(')'); }
                | '.'     { SET_TOK('.'); }
                | ':' '=' { SET_TOK(ASSIGN); }
                | '.' '.' { SET_TOK(DOTDOT); }
                | '>' '=' { SET_TOK(GE); }
                | '<' '=' { SET_TOK(LE); }
                | '<' '>' { SET_TOK(NE); } ;

an_token        : keyword | ID_;
num_token       : INT_ | REAL_ ;

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' ;
a               :  'a' | 'A' ;
b               :  'b' | 'B' ;
c               :  'c' | 'C' ;
d               :  'd' | 'D' ;
e               :  'e' | 'E' ;
f               :  'f' | 'F' ;
g               :  'g' | 'G' ;
h               :  'h' | 'H' ;
i               :  'i' | 'I' ;
j               :  'j' | 'J' ;
k               :  'k' | 'K' ;
l               :  'l' | 'L' ;
m               :  'm' | 'M' ;
n               :  'n' | 'N' ;
o               :  'o' | 'O' ;
p               :  'p' | 'P' ;
q               :  'q' | 'Q' ;
r               :  'r' | 'R' ;
s               :  's' | 'S' ;
t               :  't' | 'T' ;
u               :  'u' | 'U' ;
v               :  'v' | 'V' ;
w               :  'w' | 'W' ;
x               :  'x' | 'X' ;
y               :  'y' | 'Y' ;
z               :  'z' | 'Z' ;

most_any_char   : '\t' | '\f'
                | ' ' | '!' | '"' | '#' | '$' | '%' | '&'
                | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/'
                | '0' | '1' | '2' | '3' | '4'
                | '5' | '6' | '7' | '8' | '9'
                | ':' | ';' | '<' | '=' | '>' | '?' | '@'
                | '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'
                | '[' | '\\' | ']' | '^' | '`' | '_'
                | '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'
                | '{' | '|' | '~' ;

comment         : comment_body '}' ;
comment_body    : '{'
                | comment_body  most_any_char
                | comment_body  '\n'
                | comment_body  '\'' ;

INT_            : digit_string  { SET_TOK(INT); } ;
digit_string    : digit
                | digit_string digit ;

REAL_           : digit_string '.' digit_string
                  { SET_TOK(REAL); }
                | digit_string '.' digit_string e opt_sign digit_string
                  { SET_TOK(REAL); }
                | digit_string e opt_sign digit_string
                  { SET_TOK(REAL); } ;

opt_sign        : /* nothing */  | '+' | '-' ;

STRING_         : string_body '\''  { SET_TOK(STRING); } ;
string_body     : '\'' most_any_char
                | '\'' '}'
                | '\'' '\'' '\''
                | string_body most_any_char
                | string_body '}'
                | string_body '\'' '\'' ;

AND_            : a n d         { SET_TOK(AND); } ;
ARRAY_          : a r r a y     { SET_TOK(ARRAY); } ;
BEGINB_         : b e g i n     { SET_TOK(BEGINB); } ;
CASE_           : c a s e       { SET_TOK(CASE); } ;
CONST_          : c o n s t     { SET_TOK(CONST); } ;
DO_             : d o           { SET_TOK(DO); } ;
DOWNTO_         : DO_ w n t o   { SET_TOK(DOWNTO); } ;
ELSE_           : e l s e       { SET_TOK(ELSE); } ;
END_            : e n d         { SET_TOK(END); } ;
FOR_            : f o r         { SET_TOK(FOR); } ;
FUNCTION_       : f u n c t i o n       { SET_TOK(FUNCTION); } ;
GOTO_           : g o t o       { SET_TOK(GOTO); } ;
IDIV_           : d i v         { SET_TOK(IDIV); } ;
IF_             : i f           { SET_TOK(IF); } ;
IN_             : i n           { SET_TOK(IN); } ;
LABEL_          : l a b e l     { SET_TOK(LABEL); } ;
MOD_            : m o d         { SET_TOK(MOD); } ;
NIL_            : n i l         { SET_TOK(NIL); } ;
NOT_            : n o t         { SET_TOK(NOT); } ;
```

```
OF_             : o f           { SET_TOK(OF); } ;
OR_             : o r           { SET_TOK(OR); } ;
PACKED_         : p a c k e d   { SET_TOK(PACKED); } ;
PHYLE_          : f i l e       { SET_TOK(PHYLE); } ;
PROCEDURE_      : p r o c e d u r e     { SET_TOK(PROCEDURE); } ;
PROGRAM_        : p r o g r a m { SET_TOK(PROGRAM); } ;
RECORD_         : r e c o r d   { SET_TOK(RECORD); } ;
REPEAT_         : r e p e a t   { SET_TOK(REPEAT); } ;
SET_            : s e t         { SET_TOK(SET); } ;
THEN_           : t h e n       { SET_TOK(THEN); } ;
TO_             : t o           { SET_TOK(TO); } ;
TYPE_           : t y p e       { SET_TOK(TYPE); } ;
UNTIL_          : u n t i l     { SET_TOK(UNTIL); } ;
VAR_            : v a r         { SET_TOK(VAR); } ;
WHILE_          : w h i l e     { SET_TOK(WHILE); } ;
WITH_           : w i t h       { SET_TOK(WITH); } ;

keyword         : AND_ | ARRAY_ | BEGINB_ | CASE_ | CONST_ | DO_
                | DOWNTO_ | ELSE_ | END_ | FOR_ | FUNCTION_
                | GOTO_ | IDIV_ | IF_ | IN_ | LABEL_ | MOD_
                | NIL_ | NOT_ | OF_ | OR_ | PACKED_ | PHYLE_
                | PROCEDURE_ | PROGRAM_ | RECORD_ | REPEAT_
                | SET_ | THEN_ | TO_ | TYPE_ | UNTIL_ | VAR_
                | WHILE_ | WITH_ ;

partial_keyword : a n | a r | a r r | a r r a
                | b e | b e g | b e g i
                | c a | c a s | c o | c o n | c o n s
                | d i | DO_ w | DO_ w n | DO_ w n t
                | e l | e l s | e n | f i | f i l
                | f o | f u | f u n | f u n c | f u n c t
                | f u n c t i | f u n c t i o
                | g o | g o t | l a | l a b | l a b e
                | m o | n i | n o
                | p a | p a c | p a c k | p a c k e
                | p r | p r o | p r o c | p r o c e
                | p r o c e d | p r o c e d u
                | p r o c e d u r
                | p r o g | p r o g r | p r o g r a
                | r e | r e c | r e c o | r e c o r
                | r e p | r e p e | r e p e a
                | s e | t h | t h e | t y | t y p
                | u n | u n t | u n t i
                | v a | w h | w h i | w h i l
                | w i | w i t ;

ID_             : letter            { SET_TOK(ID); }
                | ID_ letter        { SET_TOK(ID); }
                | ID_ digit         { SET_TOK(ID); }
                | partial_keyword   { SET_TOK(ID); }
                | keyword letter    { SET_TOK(ID); }
                | keyword digit     { SET_TOK(ID); } ;
%%
```

Appendix C

A Single-Metalanguage Single-Phase Pascal Recognizer

This appendix contains the description of a single-metalanguage single-phase Pascal recognizer. This grammar was constructed by combining the two grammars of the single-metalanguage two-phase recognizer PRYY. This grammar still has all the ambiguities of the two separate grammars.

```
/*
 *  propy.y
 *  =======
 *
 *  Yacc script for a complete Pascal recognizer.
 *
 *  Difference from prly:
 *  Elipsis in a range cannot immediately follow a digit.
 */

%start program
%%
program          : program_head block DOT ;
program_head     : PROGRAM ID program_parms SEMI decls ;
program_parms    :
                 | LPAR file_list RPAR ;

file_list        : ID
                 | file_list COMMA ID ;

decls            : label_decl_part const_decl_part
                   type_decl_part var_decl_part
                   proc_decl_part ;

block            : prologue_part compound_stat
                   epilogue_part ;

prologue_part    : ;
epilogue_part    : ;

label_decl_part :
                 | LABEL label_decl_list SEMI ;

label_decl_list : label_decl
                 | label_decl_list COMMA label_decl ;

label_decl       : INT ;

const_decl_part :
                 | CONST const_decl_list SEMI ;

const_decl_list : const_decl
                 | const_decl_list SEMI const_decl ;
```

```
const_decl      : ID EQUAL const ;

const           : unsigned_num
                | PLUS unsigned_num | MINUS unsigned_num
                | ID | PLUS ID | MINUS ID | STRING ;

unsigned_num    : INT | REAL ;

type_decl_part  :
                | TYPE type_decl_list SEMI ;

type_decl_list  : type_decl
                | type_decl_list SEMI type_decl ;

type_decl       : ID EQUAL type ;

type            : simple_type | structured_type
                | POINTER ID ;

simple_type     : scalar_type | ID ;

scalar_type     : LPAR  scalar_list  RPAR
                | const DOTDOT const ;

scalar_list     : ID | scalar_list COMMA ID ;

structured_type : u_struct_type | PACKED u_struct_type ;

u_struct_type   : ARRAY LSQR   array_rest
                | RECORD field_list END
                | SET OF simple_type | PHYLE OF type ;

array_rest      : simple_type RSQR OF type
                | simple_type COMMA array_rest ;

field_list      : fixed_part | fixed_part SEMI
                | fixed_part SEMI variant_part
                | variant_part | ;

fixed_part      : fixed_item_list
                | fixed_part SEMI fixed_item_list ;

fixed_item_list : ID COLON type
                | ID COMMA fixed_item_list ;

variant_part    : CASE tag_field OF variant_list
                | CASE tag_field OF variant_list SEMI ;

tag_field       : ID | ID COLON ID ;

variant_list    : variant | variant_list SEMI variant ;

variant         : case_label_list COLON LPAR field_list
                  RPAR ;
```

```
case_label_list : const | case_label_list COMMA const ;

var_decl_part   :
                | VAR var_decl_list SEMI ;

var_decl_list   : var_decl
                | var_decl_list SEMI var_decl ;

var_decl        : ID COLON type
                | ID COMMA var_decl ;

proc_decl_part  :
                | proc_decl_list ;

proc_decl_list  : proc_decl
                | proc_decl_list proc_decl ;

proc_decl       : proc_heading block SEMI
                | proc_beg f_parm_decl SEMI ID SEMI
                | func_beg f_parm_decl COLON ID SEMI ID SEMI ;

proc_heading    : proc_head_beg decls
                | func_head_beg decls
                | func_beg SEMI decls ;

proc_head_beg   : proc_beg f_parm_decl SEMI ;
func_head_beg   : func_beg f_parm_decl COLON ID SEMI ;
proc_beg        : PROCEDURE ID ;
func_beg        : FUNCTION ID ;

f_parm_decl     : LPAR f_parm_list RPAR | ;

f_parm_list     : f_parm | f_parm_list SEMI f_parm ;

f_parm          : val_fparm_list
                | VAR var_fparm_list
                | func_beg f_parm_decl COLON ID
                | proc_beg f_parm_decl ;

val_fparm_list  : ID COLON type
                | ID COMMA val_fparm_list ;

var_fparm_list  : ID COLON type
                | ID COMMA var_fparm_list ;

compound_stat   : BEGINB stat_list END ;

stat_list       : stat | stat_list SEMI stat ;

stat            : ul_stat | label ul_stat ;

label           : INT COLON ;

ul_stat         : simple_stat | struct_stat | ;
```

```
simple_stat     : var ASSIGN expr | proc_invok
                | no_hassel_stat | GOTO INT ;

proc_invok      : noparms_pinvok
                | plist_pinvok RPAR
                | noparms_pinvok LPAR RPAR ;

noparms_pinvok  : ID ;

plist_pinvok    : noparms_pinvok LPAR parm
                | plist_pinvok COMMA parm ;

var             : ID | var DOT ID
                | subscripted_var RSQR | var POINTER ;

subscripted_var : var LSQR expr
                | subscripted_var COMMA expr ;

parm            : expr | expr COLON expr
                | expr COLON expr COLON expr ;

expr            : simple_expr | expr EQUAL simple_expr
                | expr NE simple_expr | expr LE simple_expr
                | expr LT simple_expr | expr GE simple_expr
                | expr GT simple_expr | expr IN simple_expr
                ;

simple_expr     : term | PLUS term | MINUS term
                | simple_expr PLUS term
                | simple_expr MINUS term
                | simple_expr OR term ;

term            : factor | term MULT factor
                | term DIV factor | term IDIV factor
                | term MOD factor | term AND factor ;

factor          : var | unsigned_const
                | LPAR expr RPAR
                | func_invok | set | NOT factor ;

unsigned_const  : unsigned_num | STRING | NIL ;

func_invok      : plist_finvok  RPAR
                | start_finvok LPAR RPAR ;

plist_finvok    : start_finvok LPAR parm
                | plist_finvok COMMA parm ;

start_finvok    : ID ;

set             : LSQR element_list RSQR | LSQR RSQR ;

element_list    : element | element_list COMMA element ;
```

```
element          : expr | expr DOTDOT expr ;

struct_stat      : if_then_else stat | if_beg stat
                 | while_beg DO stat | for_beg DO stat
                 | with_beg DO stat ;

if_then_else     : if_beg matched_stat ELSE ;

if_beg           : IF expr THEN ;

while_beg        : WHILE expr ;

for_beg          : for_init updown expr ;

for_init         : FOR ID ASSIGN expr ;

updown           : TO | DOWNTO ;

with_beg         : WITH rec_list ;

no_hassel_stat   : compound_stat
                 | case_alt stat SEMI END
                 | case_alt stat END
                 | repeat_beg stat_list UNTIL expr ;

repeat_beg       : REPEAT ;

case_beg         : CASE expr OF const
                 | case_beg COMMA const
                 | case_alt stat SEMI const ;

case_alt         : case_beg COLON ;

matched_stat     : ul_m_stat | label ul_m_stat ;

ul_m_stat        : simple_stat
                 |
                 | if_then_else matched_stat
                 | while_beg DO matched_stat
                 | for_beg DO matched_stat
                 | with_beg DO matched_stat
                 ;

rec_list         : var | rec_list COMMA var ;

/* ------ scanner module ------ */

optional_white   : /* nothing */ | white ;
white            : one_white  |  white one_white ;
one_white        : ' '  |  '\t'  |  '\f'  |  '\n'  | comment ;

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;
a               :  'a' | 'A' ;
b               :  'b' | 'B' ;
c               :  'c' | 'C' ;
d               :  'd' | 'D' ;
e               :  'e' | 'E' ;
f               :  'f' | 'F' ;
g               :  'g' | 'G' ;
h               :  'h' | 'H' ;
i               :  'i' | 'I' ;
j               :  'j' | 'J' ;
k               :  'k' | 'K' ;
l               :  'l' | 'L' ;
m               :  'm' | 'M' ;
n               :  'n' | 'N' ;
o               :  'o' | 'O' ;
p               :  'p' | 'P' ;
q               :  'q' | 'Q' ;
r               :  'r' | 'R' ;
s               :  's' | 'S' ;
t               :  't' | 'T' ;
u               :  'u' | 'U' ;
v               :  'v' | 'V' ;
w               :  'w' | 'W' ;
x               :  'x' | 'X' ;
y               :  'y' | 'Y' ;
z               :  'z' | 'Z' ;

most_any_char   : '\t' | '\f'
                | ' ' | '!' | '"' | '#' | '$' | '%' | '&'
                | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/'
                | ':' | ';' | '<' | '=' | '>' | '?'
                | '@' | '[' | '\\' | ']' | '^' | '`'
                | '{' | '|' | '~'
                | letter | digit ;

any_string_char : most_any_char | '}' ;
any_comment_char : most_any_char | '\n' | '\'' ;
comment         : comment_body '}' ;
comment_body    : '{'
                | comment_body  any_comment_char ;


MINUS           : MINUS_ optional_white ;
MINUS_          : '-'  ;

PLUS            : PLUS_ optional_white ;
PLUS_           : '+'  ;

MULT            : MULT_ optional_white ;
MULT_           : '*'  ;
```

```
DIV             : DIV_ optional_white ;
DIV_            : '/'  ;

EQUAL           : EQUAL_ optional_white ;
EQUAL_          : '='  ;

LT              : LT_ optional_white ;
LT_             : '<'  ;

GT              : GT_ optional_white ;
GT_             : '>'  ;

COLON           : COLON_ optional_white ;
COLON_          : ':'  ;

SEMI            : SEMI_ optional_white ;
SEMI_           : ';'  ;

COMMA           : COMMA_ optional_white ;
COMMA_          : ','  ;

LSQR            : LSQR_ optional_white ;
LSQR_           : '['  ;
                | '(' '.';

RSQR            : RSQR_ optional_white ;
RSQR_           : ']'  ;
                | '.' ')'  ;

POINTER         : POINTER_ optional_white ;
POINTER_        : '^'  ;
                | '@'  ;

LPAR            : LPAR_ optional_white ;
LPAR_           : '('  ;

RPAR            : RPAR_ optional_white ;
RPAR_           : ')'  ;

DOT             : DOT_ optional_white ;
DOT_            : '.'  ;

ASSIGN          : ASSIGN_ optional_white ;
ASSIGN_         : ':' '=' ;

DOTDOT          : DOTDOT_ optional_white ;
DOTDOT_         : '.' '.' ;

GE              : GE_ optional_white ;
GE_             : '>' '=' ;

LE              : LE_ optional_white ;
LE_             : '<' '=' ;
```

```
NE              : NE_ optional_white ;
NE_             : '<' '>' ;


INT             : INT_ optional_white ;
INT_            : digit_string  ;
digit_string    : digit | digit_string digit ;

REAL            : REAL_ optional_white ;
REAL_           : digit_string DOT_ digit_string
                | digit_string e opt_sign digit_string
                | digit_string DOT_ digit_string
                                    e opt_sign digit_string ;
opt_sign        : /* nothing */  | PLUS_ | MINUS_ ;

STRING          : STRING_ optional_white ;
STRING_         : string_body '\''  ;
string_body     : '\'' any_string_char
                | '\'' '\'' '\''
                | string_body any_string_char
                | string_body '\'' '\'' ;

AND             : AND_ optional_white ;
AND_            : a n d         ;

ARRAY           : ARRAY_ optional_white ;
ARRAY_          : a r r a y     ;

BEGINB          : BEGINB_ optional_white ;
BEGINB_         : b e g i n     ;

CASE            : CASE_ optional_white ;
CASE_           : c a s e       ;

CONST           : CONST_ optional_white ;
CONST_          : c o n s t     ;

DO              : DO_ optional_white ;
DO_             : d o           ;

DOWNTO          : DOWNTO_ optional_white ;
DOWNTO_         : DO_ w n t o   ;

ELSE            : ELSE_ optional_white ;
ELSE_           : e l s e       ;

END             : END_ optional_white ;
END_            : e n d         ;

FOR             : FOR_ optional_white ;
FOR_            : f o r         ;

FUNCTION        : FUNCTION_ optional_white ;
FUNCTION_       : f u n c t i o n       ;
```

```
GOTO            : GOTO_ optional_white ;
GOTO_           : g o t o       ;

IDIV            : IDIV_ optional_white ;
IDIV_           : d i v         ;

IF              : IF_ optional_white ;
IF_             : i f           ;

IN              : IN_ optional_white ;
IN_             : i n           ;

LABEL           : LABEL_ optional_white ;
LABEL_          : l a b e l     ;

MOD             : MOD_ optional_white ;
MOD_            : m o d         ;

NIL             : NIL_ optional_white ;
NIL_            : n i l         ;

NOT             : NOT_ optional_white ;
NOT_            : n o t         ;

OF              : OF_ optional_white ;
OF_             : o f           ;

OR              : OR_ optional_white ;
OR_             : o r           ;

PACKED          : PACKED_ optional_white ;
PACKED_         : p a c k e d   ;

PHYLE           : PHYLE_ optional_white ;
PHYLE_          : f i l e       ;

PROCEDURE       : PROCEDURE_ optional_white ;
PROCEDURE_      : p r o c e d u r e     ;

PROGRAM         : optional_white PROGRAM_ optional_white ;
PROGRAM_        : p r o g r a m ;

RECORD          : RECORD_ optional_white ;
RECORD_         : r e c o r d   ;

REPEAT          : REPEAT_ optional_white ;
REPEAT_         : r e p e a t   ;

SET             : SET_ optional_white ;
SET_            : s e t         ;

THEN            : THEN_ optional_white ;
THEN_           : t h e n       ;
```

```
TO              : TO_ optional_white ;
TO_             : t o           ;

TYPE            : TYPE_ optional_white ;
TYPE_           : t y p e       ;

UNTIL           : UNTIL_ optional_white ;
UNTIL_          : u n t i l     ;

VAR             : VAR_ optional_white ;
VAR_            : v a r         ;

WHILE           : WHILE_ optional_white ;
WHILE_          : w h i l e     ;

WITH            : WITH_ optional_white ;
WITH_           : w i t h       ;

keyword         : AND_ | ARRAY_ | BEGINB_ | CASE_ | CONST_ | DO_
                | DOWNTO_ | ELSE_ | END_ | FOR_ | FUNCTION_
                | GOTO_ | IDIV_ | IF_ | IN_ | LABEL_ | MOD_
                | NIL_ | NOT_ | OF_ | OR_ | PACKED_ | PHYLE_
                | PROCEDURE_ | PROGRAM_ | RECORD_ | REPEAT_
                | SET_ | THEN_ | TO_ | TYPE_ | UNTIL_ | VAR_
                | WHILE_ | WITH_;
partial_keyword : a n | a r | a r r | a r r a
                | b e | b e g | b e g i
                | c a | c a s | c o | c o n | c o n s
                | d i | DO_ w | DO_ w n | DO_ w n t
                | e l | e l s | e n | f i | f i l | f o
                | f u | f u n | f u n c | f u n c t
                | f u n c t i | f u n c t i o
                | g o | g o t | l a | l a b | l a b e
                | m o | n i | n o
                | p a | p a c | p a c k | p a c k e
                | p r | p r o | p r o c | p r o c e
                | p r o c e d | p r o c e d u
                | p r o c e d u r
                | p r o g | p r o g r | p r o g r a
                | r e | r e c | r e c o | r e c o r
                | r e p | r e p e | r e p e a
                | s e | t h | t h e | t y | t y p
                | u n | u n t | u n t i
                | v a | w h | w h i | w h i l
                | w i | w i t ;

ID              : ID_ optional_white ;
ID_             : letter | ID_ alpha_num
                | partial_keyword | keyword alpha_num ;
%%
```

Appendix D

An SE-SAR-NSLR(1) Grammar for ISO Pascal

This appendix contains a simple-exclusion simple-adjacency-restriction noncanonical simple LR(1) grammar for ISO Pascal. The notation used here is that new rules start in column 1 and continuations of previous rules do not. Comments start with an excalmation mark (!) and continue to the end of the line.

```
SOAP 1.1

S'              = bof Program eof

Program         = ProgramHeading SEMICOLON Block DOT

ProgramHeading  = PROGRAM IDENTIFIER ProgramParameterList

ProgramParameterList =
                | LROUND IdentifierList RROUND

! ----------------------

Block           = LabelDeclarationPart
                      ConstantDefinitionPart
                      TypeDefinitionPart
                      VariableDeclarationPart
                      ProcedureAndFunctionDeclarationPart
                      CompoundStatement

LabelDeclarationPart =
                |  LABEL LabelList SEMICOLON

LabelList       = Label
                | LabelList COMMA Label

ConstantDefinitionPart =
                | CONST ConstantDefinitionList

ConstantDefinitionList = ConstantDefinition SEMICOLON
                | ConstantDefinitionList ConstantDefinition
                      SEMICOLON

TypeDefinitionPart =
                | TYPE TypeDefinitionList

TypeDefinitionList = TypeDefinition SEMICOLON
                | TypeDefinitionList TypeDefinition
                      SEMICOLON

VariableDeclarationPart =
```

```
                | VAR VariableDeclarationList

VariableDeclarationList = VariableDeclaration SEMICOLON
                | VariableDeclarationList
                      VariableDeclaration SEMICOLON

ProcedureAndFunctionDeclarationPart =
                | ProcedureAndFunctionDeclarationList

ProcedureAndFunctionDeclarationList =
                     ProcedureOrFunctionDeclaration
                     SEMICOLON
                | ProcedureAndFunctionDeclarationList
                     ProcedureOrFunctionDeclaration
                     SEMICOLON

ProcedureOrFunctionDeclaration = ProcedureDeclaration
                | FunctionDeclaration

! ----------------------

ConstantDefinition = IDENTIFIER EQUAL Constant

TypeDefinition     = IDENTIFIER EQUAL Type

VariableDeclaration = IdentifierList COLON Type

ProcedureDeclaration = ProcedureHeading SEMICOLON Block
                | ProcedureHeading SEMICOLON IDENTIFIER

FunctionDeclaration = FunctionHeading SEMICOLON Block
                | FunctionHeading SEMICOLON IDENTIFIER

! ----------------------

ProcedureHeading = PROCEDURE IDENTIFIER FormalParameterList

FunctionHeading = FUNCTION IDENTIFIER FormalParameterList
                       COLON IDENTIFIER

FormalParameterList =
                | LROUND FormalParameterSectionList RROUND

FormalParameterSectionList = FormalParameterSection
                | FormalParameterSectionList SEMICOLON
                     FormalParameterSection

FormalParameterSection = ValueParameterSpecification
                | VariableParameterSpecification
                | ProceduralParameterSpecification
                | FunctionalParameterSpecification

! ----------------------
```

```
ValueParameterSpecification =
                  IdentifierList COLON IDENTIFIER
                | IdentifierList COLON ConformantArraySchema

VariableParameterSpecification = VAR IdentifierList COLON
                      VariableParameterTypeField

VariableParameterTypeField = IDENTIFIER
                | ConformantArraySchema

ProceduralParameterSpecification = ProcedureHeading

FunctionalParameterSpecification = FunctionHeading

ConformantArraySchema = PackedConformantArraySchema
                | UnpackedConformantArraySchema

PackedConformantArraySchema = PACKED ARRAY LSQUARE
                       IndexTypeSpecification RSQUARE
                       OF IDENTIFIER

UnpackedConformantArraySchema = ARRAY LSQUARE
                      IndexTypeSpecificationList RSQUARE
                      OF VariableParameterTypeField

IndexTypeSpecificationList = IndexTypeSpecification
                | IndexTypeSpecificationList SEMICOLON
                       IndexTypeSpecification

IndexTypeSpecification = IDENTIFIER DOTDOT IDENTIFIER COLON
                       IDENTIFIER

! ----------------------

CompoundStatement = BEGIN StatementSequence END

StatementSequence = Statement
                | StatementSequence SEMICOLON Statement

Statement       = UnlabelledStatement
                | Label COLON UnlabelledStatement

UnlabelledStatement = SimpleStatement
                | StructuredStatement

SimpleStatement =
                | AssignmentStatement
                | ProcedureStatement
                | GotoStatement

StructuredStatement = CompoundStatement
                | ConditionalStatement
                | RepetitiveStatement
                | WithStatement
```

```
ConditionalStatement = IfThen | IfThenElse | CaseStatement

RepetitiveStatement = WhileStatement | RepeatStatement
                | ForStatement

! ----------------------

AssignmentStatement = Variable ASSIGN Expression

ProcedureStatement = IDENTIFIER ActualParameterList
                | Variable

GotoStatement   = GOTO Label

IfThen          = IF Expression THEN Statement

IfThenElse      = IF Expression THEN Statement
                                ELSE Statement

IfThen -/- ELSE

CaseStatement   = CASE Expression OF CaseList END
                | CASE Expression OF CaseList SEMICOLON END

CaseList        = Case | CaseList SEMICOLON Case

RepeatStatement = REPEAT StatementSequence UNTIL Expression

WhileStatement  = WHILE Expression DO Statement

ForStatement    = FOR IDENTIFIER ASSIGN Expression
                      ToOrDownto Expression DO Statement

ToOrDownto      = TO
                | DOWNTO

WithStatement   = WITH VariableList DO Statement

VariableList    = Variable
                | Variable COMMA Variable

Case            = CaseConstantList COLON Statement

CaseConstantList = Constant
                | CaseConstantList COMMA Constant

! ----------------------

Type            = SimpleType | StructuredType
                | PointerType

SimpleType      = OrdinalType | IDENTIFIER

StructuredType  = UnpackedStructuredType
```

```
                | PACKED UnpackedStructuredType

PointerType     = POINTER DomainType

OrdinalType     = EnumeratedType | SubrangeType

UnpackedStructuredType = ArrayType | RecordType
                | SetType | FileType

DomainType      = IDENTIFIER

EnumeratedType  = LROUND IdentifierList RROUND

SubrangeType    = PreDotConstant DOTDOT Constant


ArrayType       = ARRAY LSQUARE IndexTypeList RSQUARE
                      OF ComponentType

IndexTypeList   = SimpleType
                | IndexTypeList COMMA SimpleType

RecordType      = RECORD FieldList END

SetType         = SET OF SimpleType

FileType        = FILE OF ComponentType

ComponentType   = Type

FieldList       =
                | FixedPart | FixedPart SEMICOLON
                | FixedPart SEMICOLON VariantPart
                | FixedPart SEMICOLON VariantPart SEMICOLON
                | VariantPart | VariantPart SEMICOLON

FixedPart       = RecordSection
                | FixedPart SEMICOLON RecordSection

VariantPart     = CASE VariantSelector OF Variant
                | VariantPart SEMICOLON Variant

RecordSection   = IdentifierList COLON Type

VariantSelector = TagType
                | TagField COLON TagType

Variant         = CaseConstantList COLON LROUND
                      FieldList RROUND

TagType         = IDENTIFIER

TagField        = IDENTIFIER
```

```
! ------------------------

Constant        = IDENTIFIER
                | UnsignedNumber
                | Sign IDENTIFIER
                | Sign UnsignedNumber
                | CharacterString

! Special constant to eliminate SLR inconsistency.
PreDotConstant  = IDENTIFIER
                | UnsignedNumber
                | Sign IDENTIFIER
                | Sign UnsignedNumber
                | CharacterString

! ------------------------

Expression      = SimpleExpression
                | SimpleExpression
                      RelationalOperator SimpleExpression

SimpleExpression = UnsignedSimpleExpression
                | Sign UnsignedSimpleExpression

UnsignedSimpleExpression = Term
                | UnsignedSimpleExpression
                      AddingOperator Term

Term            = Factor
                | Term MultiplyingOperator Factor

Factor          = UnsignedConstant | Variable
                | SetConstructor
                | IDENTIFIER ActualParameterList
                | NOT Factor
                | LROUND Expression RROUND

RelationalOperator = EQUAL | NE | LT | LE | GT | GE | IN

AddingOperator  = PLUS | MINUS | OR

MultiplyingOperator = STAR | SLASH | DIV | MOD | AND

UnsignedConstant = UnsignedNumber | CharacterString | NIL

! ------------------------


Variable        = IDENTIFIER
                | Variable DOT IDENTIFIER
                | Variable POINTER
                | Variable LSQUARE IndexList RSQUARE

IndexList       = Expression
```

```
                | IndexList COMMA Expression

SetConstructor  = LSQUARE RSQUARE
                | LSQUARE ElementDescriptionList RSQUARE

ElementDescriptionList = ElementDescription
                | ElementDescriptionList COMMA
                      ElementDescription

ElementDescription = Expression
                | Expression DOTDOT Expression

ActualParameterList = LROUND ActualParameterPart RROUND

ActualParameterPart = ActualParameter
                | ActualParameterPart COMMA ActualParameter

ActualParameter = Expression
                | Expression COLON Expression
                | Expression COLON Expression
                             COLON Expression

! ----------------------

UnsignedNumber  = UnsignedInteger White
                | UnsignedReal White

IdentifierList  = IDENTIFIER
                | IdentifierList COMMA IDENTIFIER

Label           = DigitSequence White

Sign            = PLUS | MINUS

! ----------------------
!  Indivisible symbols
! ----------------------

IDENTIFIER      = Identifier White

Identifier      = IdentifierFragment

IdentifierFragment = Letter
                | IdentifierFragment Letter
                | IdentifierFragment Digit

UnsignedInteger = DigitSequence

UnsignedReal    = DigitSequence DOT DigitSequence
                | DigitSequence DOT DigitSequence
                      e ScaleFactor
                | DigitSequence e ScaleFactor

ScaleFactor     = DigitSequence
```

```
                | "+" DigitSequence
                | "-" DigitSequence

CharacterString = "'" StringElementSequence "'" White

StringElementSequence = StringElement
                | StringElementSequence StringElement

DigitSequence   = Digit
                | DigitSequence Digit

Comment         = LCURLY CommentElementSequence "}"
                | LCURLY CommentElementSequence "*)"

CommentElementSequence = CommentElement
                | CommentElementSequence CommentElement


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"

StringElement   = "''"
                | StringOrCommentElement
                | "*" | "}"

CommentElement  = StringOrCommentElement
                | "'" | StarNoRRound | "\n"

StarNoRRound    = "*"

StringOrCommentElement  = "\t" | " " | "\f" | "\\" | "!"
                | "\"" | "#" | "$" | "%" | "&" | "(" | ")"
                | "+" | "," | "-" | "." | "/" | "0" | "1"
                | "2" | "3" | "4" | "5" | "6" | "7" | "8"
                | "9" | ":" | ";" | "<" | "=" | ">" | "?"
                | "@" | "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" | "["
                | "]" | "^" | "_" | "`" | "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" | "{" | "|" | "~"
!               | "\n"          ! Deleted for strings.
!               | "'"           ! Deleted for strings.
!               | "*"           ! Deleted for comments.
!               | "}"           ! Deleted for comments.

!
```

```
! Punctuation
! -----------

ASSIGN          = ":="       White
SEMICOLON       = ";"    White
DOT             = "."    White
DOTDOT          = ".."       White
LROUND          = "("    White
RROUND          = ")"    White
LSQUARE         = "["    White | "(." White
RSQUARE         = "]"    White | ".)" White
LCURLY          = "{" | "(*"
COMMA           = ","    White
EQUAL           = "="    White
COLON           = ":"    White
POINTER         = "^"    White | "@"      White
PLUS            = "+"    White
MINUS           = "-"    White
NE              = "<>"       White
LT              = "<"    White
LE              = "<="       White
GT              = ">"    White
GE              = ">="       White
STAR            = "*"    White
SLASH           = "/"    White

! Reserved keywords
! -----------------

AND             = and    White
ARRAY           = array White
BEGIN           = begin White
CASE            = case   White
CONST           = const White
DIV             = div    White
DO              = do     White
DOWNTO          = downto         White
ELSE            = else   White
END             = end    White
FILE            = file   White
FOR             = for    White
FUNCTION        = function       White
GOTO            = goto   White
IF              = if     White
IN              = in     White
LABEL           = label White
MOD             = mod    White
NIL             = nil    White
NOT             = not    White
OF              = of     White
OR              = or     White
PACKED          = packed         White
PROCEDURE       = procedure      White
PROGRAM         = White  program         White
```

```
RECORD          = record        White
REPEAT          = repeat        White
SET             = set   White
THEN            = then  White
TO              = to    White
TYPE            = type  White
UNTIL           = until White
VAR             = var   White
WHILE           = while White
WITH            = with  White

and             = a n d
array           = a r r a y
begin           = b e g i n
case            = c a s e
const           = c o n s t
div             = d i v
do              = d o
downto          = d o w n t o
else            = e l s e
end             = e n d
file            = f i l e
for             = f o r
function        = f u n c t i o n
goto            = g o t o
if              = i f
in              = i n
label           = l a b e l
mod             = m o d
nil             = n i l
not             = n o t
of              = o f
or              = o r
packed          = p a c k e d
procedure       = p r o c e d u r e
program         = p r o g r a m
record          = r e c o r d
repeat          = r e p e a t
set             = s e t
then            = t h e n
to              = t o
type            = t y p e
until           = u n t i l
var             = v a r
while           = w h i l e
with            = w i t h

a               = "a" | "A"
b               = "b" | "B"
c               = "c" | "C"
d               = "d" | "D"
e               = "e" | "E"
f               = "f" | "F"
g               = "g" | "G"
```

```
h               = "h" | "H"
i               = "i" | "I"
j               = "j" | "J"
k               = "k" | "K"
l               = "l" | "L"
m               = "m" | "M"
n               = "n" | "N"
o               = "o" | "O"
p               = "p" | "P"
q               = "q" | "Q"
r               = "r" | "R"
s               = "s" | "S"
t               = "t" | "T"
u               = "u" | "U"
v               = "v" | "V"
w               = "w" | "W"
x               = "x" | "X"
y               = "y" | "Y"
z               = "z" | "Z"

!  Define white space.
!  -------------------

White           =
                | White WhiteElement

WhiteElement    = " " | "\t" | "\n" | "\f" | Comment

!
!  Disambiguation section
!  ----------------------

Identifier and array begin case const div do downto
      else end file for function goto if in label mod nil
      not of or packed procedure program record repeat
      set then to type until var while with
  -/- Identifier and array begin case const div do downto
      else end file for function goto if in label mod nil
      not of or packed procedure program record repeat
      set then to type until var while with DigitSequence

StarNoRRound -/- ")"

Identifier   # and | array | begin | case | const | div
             | do | downto | else | end | file | for
             | function | goto | if | in | label | mod
             | nil | not | of | or | packed | procedure
             | program | record | repeat | set | then
             | to | type | until | var | while | with
```

Appendix E

SE-SAR-NSLR(1) Grammars for a Two-Phase ISO Pascal Recognizer

This appendix contains two simple-exclusion simple-adjacency-restriction noncanonical simple LR(1) grammars for a two-phase ISO Pascal recognizer. The grammar for the scanner is given first, and the grammar for the parser follows it. The notation used here is that new rules start in column 1 and continuations of previous rules do not. Comments start with an excalmation mark (!) and continue to the end of the line.

E.1. The Grammar for the Scanner

```
S'              = bof White TokenSequence eof

TokenSequence   = Token White
                | TokenSequence Token White

Token           = UnsignedInteger | UnsignedReal
                | IDENTIFIER | CharacterString | ASSIGN
                | SEMICOLON | DOT | DOTDOT | LROUND
                | RROUND | LSQUARE | RSQUARE | COMMA
                | EQUAL | COLON | POINTER | PLUS | MINUS
                | NE | LT | LE | GT | GE | STAR | SLASH
                | and | array | begin | case | const
                | div | do | downto | else | end | file
                | for | function | goto | if | in
                | label | mod | nil | not | of | or
                | packed | procedure | program | record
                | repeat | set | then | to | type | until
                | var | while | with

! ----------------------
!  Indivisible symbols
! ----------------------

UnsignedInteger = DigitSequence

IDENTIFIER      = IdentifierFragment

IdentifierFragment = Letter
                | IdentifierFragment Letter
                | IdentifierFragment Digit

UnsignedReal    = DigitSequence PERIOD DigitSequence
                | DigitSequence PERIOD DigitSequence
                      ScaleFactor
                | DigitSequence ScaleFactor
```

```
ScaleFactor      = e DigitSequence
                 | e PLUS DigitSequence
                 | e MINUS DigitSequence

CharacterString = "'" StringElementSequence "'"

StringElementSequence = StringElement
                 | StringElementSequence StringElement

DigitSequence    = NumFrag

NumFrag          = Digit | NumFrag Digit

Comment          = LCURLY CommentElementSequence "}"
                 | LCURLY CommentElementSequence "*" ")"

CommentElementSequence = CommentElement
                 | CommentElementSequence CommentElement

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"

StringElement    = "'" "'" | StringOrCommentElement
                 | "*" | "}"

CommentElement   = StringOrCommentElement
                 | "'" | StarNoRRound | "\n"

StarNoRRound     = "*"       ! Star not followed by ")".

StringOrCommentElement  = "\t" | " " | "\f" | "\\" | "!"
                 | "\"" | "#" | "$" | "%" | "&" | "(" | ")"
                 | "+" | "," | "-" | "." | "/" | "0" | "1"
                 | "2" | "3" | "4" | "5" | "6" | "7" | "8"
                 | "9" | ":" | ";" | "<" | "=" | ">" | "?"
                 | "@" | "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" | "["
                 | "]" | "^" | "_" | "`" | "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" | "{" | "|" | "~"
     !           | "\n" | "'"          ! Deleted for strings.
     !           | "*" | "}"           ! Deleted for comments.

!
! Punctuation
! -----------
```

```
ASSIGN          = ":" "="
SEMICOLON       = ";"
PERIOD          = "."
DOT             = "."
DOTDOT          = "." "."
LROUND          = "("
RROUND          = ")"
LSQUARE         = "[" | "(" "."
RSQUARE         = "]" | "." ")"
LCURLY          = "{" | "(" "*"
COMMA           = ","
EQUAL           = "="
COLON           = ":"
POINTER         = "^" | "@"
PLUS            = "+"
MINUS           = "-"
NE              = "<" ">"
LT              = "<"
LE              = "<" "="
GT              = ">"
GE              = ">" "="
STAR            = "*"
SLASH           = "/"

! Reserved keywords
! -----------------

and             = a n d
array           = a r r a y
begin           = b e g i n
case            = c a s e
const           = c o n s t
div             = d i v
do              = d o
downto          = d o w n t o
else            = e l s e
end             = e n d
file            = f i l e
for             = f o r
function        = f u n c t i o n
goto            = g o t o
if              = i f
in              = i n
label           = l a b e l
mod             = m o d
nil             = n i l
not             = n o t
of              = o f
or              = o r
packed          = p a c k e d
procedure       = p r o c e d u r e
program         = p r o g r a m
record          = r e c o r d
repeat          = r e p e a t
```

```
set             = s e t
then            = t h e n
to              = t o
type            = t y p e
until           = u n t i l
var             = v a r
while           = w h i l e
with            = w i t h

a               = "a" | "A"
b               = "b" | "B"
c               = "c" | "C"
d               = "d" | "D"
e               = "e" | "E"
f               = "f" | "F"
g               = "g" | "G"
h               = "h" | "H"
i               = "i" | "I"
j               = "j" | "J"
k               = "k" | "K"
l               = "l" | "L"
m               = "m" | "M"
n               = "n" | "N"
o               = "o" | "O"
p               = "p" | "P"
q               = "q" | "Q"
r               = "r" | "R"
s               = "s" | "S"
t               = "t" | "T"
u               = "u" | "U"
v               = "v" | "V"
w               = "w" | "W"
x               = "x" | "X"
y               = "y" | "Y"
z               = "z" | "Z"

!  Define white space.
!  -------------------

White           =
                | White WhiteElement

WhiteElement    = " " | "\t" | "\f" | "\n" | Comment

!
!  Disambiguation section
!  ----------------------

IDENTIFIER and array begin case const div do downto
      else end file for function goto if in label mod
      nil not of or packed procedure program record
      repeat set then to type until var while with
  -/- IDENTIFIER and array begin case const div do downto
      else end file for function goto if in label mod
```

```
      nil not of or packed procedure program record
      repeat set then to type until var while with
      DigitSequence UnsignedInteger UnsignedReal ScaleFactor

DigitSequence -/- DigitSequence

StarNoRRound -/- ")"

UnsignedReal UnsignedInteger
  -/- UnsignedInteger UnsignedReal ScaleFactor e

CharacterString -/- CharacterString

LROUND -/- STAR DOT DOTDOT RSQUARE

LSQUARE -/- RROUND

DOTDOT -/- RROUND DOT DOTDOT RSQUARE

DOT -/- RROUND DOT DOTDOT RSQUARE UnsignedInteger
           UnsignedReal

PERIOD -/- RROUND DOT DOTDOT RSQUARE

COLON -/- EQUAL

EQUAL NE LT LE GT GE -/- EQUAL NE LT LE GT GE

IDENTIFIER   # and | array | begin | case | const | div
             | do | downto | else | end | file | for
             | function | goto | if | in | label | mod
             | nil | not | of | or | packed | procedure
             | program | record | repeat | set | then
             | to | type | until | var | while | with
```

E.2. The Grammar for the Parser

```
S'              = bof Program eof

Program         = ProgramHeading SEMICOLON Block DOT

ProgramHeading  = PROGRAM IDENTIFIER ProgramParameterList

ProgramParameterList =
                | LROUND IdentifierList RROUND


! ----------------------

Block           = LabelDeclarationPart
                ConstantDefinitionPart
                TypeDefinitionPart
```

```
                VariableDeclarationPart
                ProcedureAndFunctionDeclarationPart
                CompoundStatement

LabelDeclarationPart =
                |  LABEL LabelList SEMICOLON

LabelList       = DigitSequence
                | LabelList COMMA DigitSequence

ConstantDefinitionPart =
                | CONST ConstantDefinitionList

ConstantDefinitionList = ConstantDefinition SEMICOLON
                | ConstantDefinitionList ConstantDefinition
                      SEMICOLON

TypeDefinitionPart =
                | TYPE TypeDefinitionList

TypeDefinitionList = TypeDefinition SEMICOLON
                | TypeDefinitionList TypeDefinition
                      SEMICOLON

VariableDeclarationPart =
                | VAR VariableDeclarationList

VariableDeclarationList = VariableDeclaration SEMICOLON
                | VariableDeclarationList
                      VariableDeclaration SEMICOLON

ProcedureAndFunctionDeclarationPart =
                | ProcedureAndFunctionDeclarationList

ProcedureAndFunctionDeclarationList =
                ProcedureOrFunctionDeclaration SEMICOLON
          | ProcedureAndFunctionDeclarationList
                ProcedureOrFunctionDeclaration SEMICOLON

ProcedureOrFunctionDeclaration = ProcedureDeclaration
          | FunctionDeclaration

! ----------------------

ConstantDefinition = IDENTIFIER EQUAL Constant

TypeDefinition  = IDENTIFIER EQUAL Type

VariableDeclaration = IdentifierList COLON Type

ProcedureDeclaration = ProcedureHeading SEMICOLON Block
                | ProcedureHeading SEMICOLON IDENTIFIER

FunctionDeclaration = FunctionHeading SEMICOLON Block
```

```
                | FunctionHeading SEMICOLON IDENTIFIER

! ----------------------

ProcedureHeading = PROCEDURE IDENTIFIER FormalParameterList

FunctionHeading = FUNCTION IDENTIFIER FormalParameterList
                              COLON IDENTIFIER

FormalParameterList =
                | LROUND FormalParameterSectionList RROUND

FormalParameterSectionList = FormalParameterSection
                | FormalParameterSectionList SEMICOLON
                     FormalParameterSection

FormalParameterSection = ValueParameterSpecification
                | VariableParameterSpecification
                | ProceduralParameterSpecification
                | FunctionalParameterSpecification

! ----------------------

ValueParameterSpecification =
               IdentifierList COLON IDENTIFIER
                | IdentifierList COLON ConformantArraySchema

VariableParameterSpecification = VAR IdentifierList COLON
                      VariableParameterTypeField

VariableParameterTypeField = IDENTIFIER
         | ConformantArraySchema

ProceduralParameterSpecification = ProcedureHeading

FunctionalParameterSpecification = FunctionHeading

ConformantArraySchema = PackedConformantArraySchema
                | UnpackedConformantArraySchema

PackedConformantArraySchema = PACKED ARRAY LSQUARE
                      IndexTypeSpecification RSQUARE
                      OF IDENTIFIER

UnpackedConformantArraySchema = ARRAY LSQUARE
                    IndexTypeSpecificationList RSQUARE
                    OF VariableParameterTypeField

IndexTypeSpecificationList = IndexTypeSpecification
         | IndexTypeSpecificationList SEMICOLON
               IndexTypeSpecification

IndexTypeSpecification = IDENTIFIER DOTDOT IDENTIFIER COLON
                              IDENTIFIER
```

```
! ----------------------


CompoundStatement = BEGIN StatementSequence END

StatementSequence = Statement
                  | StatementSequence SEMICOLON Statement

Statement         = UnlabelledStatement
                  | Label COLON UnlabelledStatement

UnlabelledStatement = SimpleStatement | StructuredStatement

SimpleStatement =
                  | AssignmentStatement | ProcedureStatement
                  | GotoStatement

StructuredStatement = CompoundStatement
                  | ConditionalStatement
                  | RepetitiveStatement | WithStatement

ConditionalStatement = IfStatement | CaseStatement

RepetitiveStatement = WhileStatement | RepeatStatement
                  | ForStatement

! ----------------------

AssignmentStatement = Variable ASSIGN Expression

ProcedureStatement = IDENTIFIER ActualParameterList
                  | Variable

GotoStatement     = GOTO Label

IfStatement       = IfThen
                  | IfThenElse

IfThen            = IF Expression THEN Statement

IfThenElse        = IF Expression THEN Statement
                                ELSE Statement

IfThen -/- ELSE

CaseStatement     = CASE Expression OF CaseList END
                  | CASE Expression OF CaseList SEMICOLON END

CaseList          = Case
                  | CaseList SEMICOLON Case

RepeatStatement = REPEAT StatementSequence UNTIL Expression

WhileStatement  = WHILE Expression DO Statement
```

```
ForStatement     = FOR IDENTIFIER ASSIGN Expression
                        ToOrDownto Expression DO Statement

ToOrDownto       = TO | DOWNTO

WithStatement    = WITH VariableList DO Statement

VariableList     = Variable
                 | Variable COMMA Variable

Case             = CaseConstantList COLON Statement

CaseConstantList = Constant
                 | CaseConstantList COMMA Constant

! ----------------------

Type             = SimpleType | StructuredType | PointerType

SimpleType       = OrdinalType | IDENTIFIER

StructuredType   = UnpackedStructuredType
                 | PACKED UnpackedStructuredType

PointerType      = POINTER DomainType

OrdinalType      = EnumeratedType | SubrangeType

UnpackedStructuredType = ArrayType | RecordType | SetType
                 | FileType

DomainType       = IDENTIFIER

EnumeratedType   = LROUND IdentifierList RROUND

SubrangeType     = PreDotConstant DOTDOT Constant

ArrayType        = ARRAY LSQUARE IndexTypeList RSQUARE
                        OF ComponentType

IndexTypeList    = SimpleType
                 | IndexTypeList COMMA SimpleType

RecordType       = RECORD FieldList END

SetType          = SET OF SimpleType

FileType         = FILE OF ComponentType

ComponentType    = Type

FieldList        =
                 | FixedPart | FixedPart SEMICOLON
                 | FixedPart SEMICOLON VariantPart
```

```
                | FixedPart SEMICOLON VariantPart SEMICOLON
                | VariantPart | VariantPart SEMICOLON

FixedPart       = RecordSection
                | FixedPart SEMICOLON RecordSection

VariantPart     = CASE VariantSelector OF Variant
                | VariantPart SEMICOLON Variant

RecordSection   = IdentifierList COLON Type

VariantSelector = TagType
                | TagField COLON TagType

Variant         = CaseConstantList COLON LROUND
                      FieldList RROUND

TagType         = IDENTIFIER

TagField        = IDENTIFIER

! ----------------------

Constant        = IDENTIFIER | UnsignedNumber
                | Sign IDENTIFIER | Sign UnsignedNumber
                | CharacterString

! Special constant to eliminate SLR inconsistency.
PreDotConstant  = IDENTIFIER | UnsignedNumber
                | Sign IDENTIFIER | Sign UnsignedNumber
                | CharacterString

! ----------------------

Expression      = SimpleExpression
                | SimpleExpression RelationalOperator
                      SimpleExpression

SimpleExpression = UnsignedSimpleExpression
                | Sign UnsignedSimpleExpression

UnsignedSimpleExpression = Term
                | UnsignedSimpleExpression
                      AddingOperator Term

Term            = Factor
                | Term MultiplyingOperator Factor

Factor          = UnsignedConstant | Variable
                | SetConstructor
                | IDENTIFIER ActualParameterList
                | NOT Factor | LROUND Expression RROUND

RelationalOperator = EQUAL | NE | LT | LE | GT | GE | IN
```

```
AddingOperator  = PLUS | MINUS | OR

MultiplyingOperator = STAR | SLASH | DIV | MOD | AND

UnsignedConstant = UnsignedNumber | CharacterString | NIL

! ----------------------

Variable        = IDENTIFIER | Variable DOT IDENTIFIER
                | Variable POINTER
                | Variable LSQUARE IndexList RSQUARE

IndexList       = Expression | IndexList COMMA Expression

SetConstructor  = LSQUARE RSQUARE
                | LSQUARE ElementDescriptionList RSQUARE

ElementDescriptionList = ElementDescription
                | ElementDescriptionList COMMA
                      ElementDescription

ElementDescription = Expression
                | Expression DOTDOT Expression

ActualParameterList = LROUND ActualParameterPart RROUND

ActualParameterPart = ActualParameter
                | ActualParameterPart COMMA ActualParameter

ActualParameter = Expression
                | Expression COLON Expression
                | Expression COLON Expression
                             COLON Expression

! ----------------------

UnsignedNumber  = UnsignedInteger | UnsignedReal

IdentifierList  = IDENTIFIER
                | IdentifierList COMMA IDENTIFIER

Label           = UnsignedInteger

Sign            = PLUS
                | MINUS
```

Appendix F

SOAP: An SE-SAR-NSLR(1) Parser Generator

User's Guide

Daniel J. Salomon

F.1. Introduction

This document describes the use of the SE-SAR-NSLR(1) parser generator called SOAP. The name "SOAP" was devised by rearranging the first letters of the words "one-phase syntax analysis". This program reads a restricted context-free grammar and builds the states and lookahead sets necessary to implement a noncanonical shift-reduce parser. Two forms of output are produced. The first is a file containing a human-readable listing of the vocabulary, grammar, and state sets. The second is a file that contains similar information but is designed to facilitate further machine processing.

F.2. Input Format

The input to SOAP is a restricted context-free grammar. For parser generation to be completed correctly, the input grammar must be in the class SE-SAR-NSLR(1), i.e. simple-exclusion simple-adjacency-restriction noncanonical SLR(1).

F.2.1. Comments and Blank Lines

Comments may be inserted anywhere in the input file. Comments start with an exclamation mark "!", and continue to the end of the line. Blank lines may be inserted anywhere in the input file.

F.2.2. Grammar Symbols

Terminals and nonterminals may be almost any sequence of characters delimited by blanks, tabs, formfeeds, or newlines. This convention allows punctuation marks to be used as symbol names, and thus allows very concise grammars. It also allows very descriptive symbol names such as "(letter|digit)*".

Some strings, however, have a special meaning and cannot be used alone as symbol names. They are:

= – (equal sign),
| – (alternation bar),
@ – (at sign),
# – (number sign), and
-/- – (adjacency-restriction operator).

The meanings of these special symbols are explained in later sections. In addition the symbols "<BOF>" and "<EOF>" are predefined terminal symbols reserved for representing beginning-of-file and end-of-file, respectively. These strings can be used as part of symbol names, but not alone as symbol names, thus a common technique is to escape them with a backslash. Thus \= is a valid symbol name even though = is not.

Symbol names are case sensitive, so that "IDENT" and "Ident" are different symbol names.

As its name implies, SOAP is intended for writing scannerless parsers of programming languages, and hence special treatment is given for symbols intended to represent input characters. To specify terminal symbols that represent a single input character, enclose that character in quotation marks. For instance the symbol "x" represents the single input character x. Escape sequences are

available for representing unprintable or syntactically troublesome characters:

\n – Newline
\f – Formfeed
\t – Horizontal tab
\" – Quotation mark
\\ – Backslash
\s – Space

A quoted string containing more than one character will be treated as a sequence of quoted symbols containing only one character each. For instance the quoted symbol "begin" is equivalent to the five symbols "b", "e", "g", "i", and "n", but is more convenient to write. A space can be embedded inside a quoted symbol, and it will be converted into the symbol "\s".

F.2.3. Grammar Rules

There are three kinds of grammar rules: productions, exclusion rules, and adjacency restriction rules. In addition every grammar must start with a line specifying the the grammar format being used. The form of each rule is presented in following sections. Each rule begins in column 1 of the input, and any line for which column 1 is a blank or tab is a continuation of the previous rule.

F.2.4. Grammar Format Specification

Every grammar must start with a line identifying the grammar format being used. The purpose of this format specification is to allow an orderly transition to new grammar formats. If the format of the grammar expected changes in the future, old grammars will not be mistakenly interpreted as being in the new format. If possible, a translator will be provided to convert old grammars to the new format.

A grammar format specification consists of a grammar version name followed by a grammar version number. Currently the format expected is:

SOAP 1

this line need not start in column one, and need not be in upper case.

F.2.5. Productions

A production takes the form:

nonterminal = right-part | alternate-right-part
| alternate-right-part | ...

Each nonterminal may appear only once as the left part of a production, but it may have as many alternate right parts as desired. SOAP assigns symbol numbers to each terminal and nonterminal symbol. A symbol that does not appear as the left part of any rule is assumed to be a terminal symbol.

The first production of the grammar should have the form:

start-symbol = <BOF> string-of-symbols <EOF>

This rule defines the distinguished start symbol. The distinguished start symbol may not be used in the right part of any rule, and may not have any alternate right parts. The predefined symbols "<BOF>" and "<EOF>" are terminals and represent "beginning of file" and "end of file" respectively.

F.2.6. Exclusion Rules

Exclusion rules have a form similar to productions, but use the operator "#" where "=" would appear. A sample exclusion rule would be:

symbol-0 # symbol-1 | symbol-2 | symbol-3 | ...

The right part and alternate right parts must consist of a single symbol. All the symbols must be

nonterminals. Currently SOAP accepts only one such construct, and it is intended to define the set of reserved keywords in a programming language.

F.2.7. Adjacency-Restriction Rules

Adjacency-restriction rules take the form:

pred-1 pred-2 pred-3 ... –/– succ-1 succ-2 succ-3 ...

Currently SOAP requires that a symbol may appear in the left part of only one adjacency restriction.

F.2.8. Semantic Actions

The names of semantic actions can be attached to grammar productions. The purpose of this feature is (1) to facilitate the binding of semantic actions to grammar rules by grammar table postprocessors, (2) to allow future versions of SOAP to generate complete parsers, rather than just parse tables, and (3) to allow vertical parse-table compression. Vertical compression is described in a section by that name, below.

A semantic-action name may follow a production right part, or alternate right part and is separated from the right part by an "@" (at sign). For example:

```
Block = BEGIN Statements END    @ process_block
      | BEGIN END               @ null_block
```

The name of the semantic action is intended to be the name of a macro or procedure that will be invoked whenever a reduction by that rule is made. A probable form of the macro or procedure would be to have one parameter for each symbol in the right part of the production to accept a pointer to an attribute record, and produce a pointer to an attribute record as a result. Rules without a semantic action are considered to perform some default semantic action called "Default". The default semantic action would probably be to copy the attribute record of the first symbol in the right part of the production and assign it to the symbol in the left part of the production.

If a semantic action is also followed by an "@" then it also applies to all the subsequent alternate rules for the current left part, or until another semantic action is specified. This saves repetition of the semantic action for long lists of highly similar rules, as are common in character-level grammars. To reselect the default semantic action after a repeated semantic action select the action "@ Default".

F.3. Parse-Table Compression

Three kinds of parse table compression are provided: reduce-action pruning, vertical compression, and horizontal compression.

F.3.1. Reduce-Action Pruning

SOAP uses the set FOLLOW(A) for lookahead symbols on reduce actions to the symbol A. This strategy is correct, but often generates many reduce actions for lookahead symbols that could not possibly appear on the lookahead stack for particular states. SOAP has some simple strategies for removing reduce actions on impossible lookahead symbols. These strategies are described elsewhere [Daniel J. Salomon, "Metalanguage Enhancements and Parser-Generation Techniques for Scannerless Parsing of Programming Languages", Ph.D. Thesis, University of Waterloo, 1989, section 4.10].

F.3.2. Vertical Compression

Vertical parse-table compression is so called because it reduces the number of parser states (number of rows in the parse table). It can be selected by the option "-v" on the command line.

The first step of vertical compression is to determine equivalent reduce actions in the parse table. Two reduce actions "REDUCE i" and "REDUCE j" are considered to be equivalent if the productions "i" and "j" by which they perform the reduction

(1) have the same left part,
(2) have the same number of symbols in the right part, and
(3) specify the same semantic action.

Equivalent reduce actions in the parse table are identified and replaced by a common reduce action. After reduce actions have been remapped, there may be identical rows in the parse table. Any parse-table row that has appeared earlier in the table is removed, and all shift transitions are remapped to reflect the new state numbers.

This kind of compression is most effective on grammars with repetitive rules such as:

Letter = a | b | c | d | ...

F.3.3. Horizontal Compression

Horizontal parse-table compression is so called because it reduces the width (number of columns) of the parse table. It can be selected by the option "-h" on the command line. If a column is identical to a previous column, it is deleted. When horizontal compression is specified, a map is prepared for mapping each lookahead symbol to the correct parser-action column.

Parser_action_column = lookmap [lookahead_symbol]

If both vertical and horizontal compression are requested, horizontal compression is applied after vertical compression. In this way it may benefit from the remapping of reduce actions that precedes vertical compression. Note, however, that reduce actions are not remapped unless vertical compression is requested.

For backward compatibility with earlier versions of the parse-table generator, the number of columns in the compressed parser-action table is not reported in the ".tbl" file. A parser that uses the compressed tables should allocate a table that has one column for each symbol. In such a table some of those columns may be unused. Horizontal compression is principally intended to be a compression of the reporting of the table.

F.4. Executing SOAP

SOAP is invoked with a command line in the following form:

soap [-lcknsduvhWwa] [input-file]

Options:

-l Inhibit production of a listing (.lst) file, produce only a parse table (.tbl) file.

-c Print closure item-set when reporting states in the listing file. (Default option.)

-k Print only kernel items when reporting states in the listing file.

-n Perform noncanonical state expansion if a state is not SLR(1) consistent. (Default option.)

-s Prepare only SLR(1) tables. Do not invoke state expansion even if state is not SLR(1) consistent.

-d Implementor debug mode. Internal tables are initialized to zero to simplify debugging, despite the extra execution time required. This option is not useful for the user.

-u Do not perform reduce-action pruning. All reduce actions are left in the parse table, even if it is known that the specified lookahead symbol cannot appear on the lookahead stack for a particular state.

-i Keep invisible symbols in lookahead sets. This option is useful only for experimenting with special parser generation techniques.

-v Perform vertical compression of parse tables.

-h Perform horizontal compression of the parse tables.

-W Ignore keyword disambiguation (exclusion) rules included in grammar.

-w Process keyword disambiguation (exclusion) rules, but keywords (the excluded symbols) are reserved only if they appear in a valid context for that keyword, otherwise they are interpreted as identifiers (the exclusion symbol). If this option is used then a semantic action should test each identifier against the list of reserved words. This option leads to smaller parse tables, at the expense of user testing for reserved words.

-a Ignore adjacency-restriction rules.

F.5. The Listing File

SOAP produces a listing file describing the parser generated. The listing file will have the same name as the input file with the file type changed to ".lst". If input comes from standard input then the listing will be produced on standard output. The generation of a listing file can be suppressed by using the option "-l".

The listing file begins by with a list of the symbols of the grammar. Each symbol is preceded by its assigned number, and by a flag of "T" or "NT" to indicate whether it is a terminal or a nonterminal.

Next the grammar is listed. The grammar rules are numbered starting from 0. If a grammar rule had a semantic action attached, the name of that action will be shown. Productions added to implement the exclusion rules in the grammar are listed with the operator "#" in place of "=". A listing of the exclusion rules and adjacency-restriction rules follows the productions.

Next the states of the SLR(1) parser are listed. Each item in each state is preceded by a flag to indicate its type:

K-- The item is a kernel item.

-C- The item was added by item-set closure on the kernel set.

--N The item was added by noncanonical state expansion.

--E The item is an epsilon item added by noncanonical state expansion.

The items themselves are listed in two forms. The first form is a numeric pair (rule, position), that gives the rule number of the item, and the position of the item dot. The second form lists the rule symbolically with a period representing the item dot inserted between the symbols of the rule. The position of the item dot shows how many symbols from the beginning of the rule have been recognized by the current state. Complete items, those items (with the item dot at the end) that lead to reduce actions, are followed by their lookahead set enclosed in braces. Incomplete items, those items that lead to shift transitions are followed by the state number of the transition destination.

F.5.1. Conflict Reporting

If any conflicts are detected, they are listed preceding the inadequate state. SLR conflicts are preceded by the marker "++++", and unresolvable conflicts are preceded by the marker ">>>>", so that they can be easily located using a text editor. The type of the conflict, SLR, NSLR, SE, or SAR, is given following the conflict marker along with a report of the items involved in the conflict. If noncanonical parse-table generation is selected, then some of the SLR conflicts may be resolved, and the unresolved ones will be reported as NSLR conflicts.

F.5.1.1. Simple-Exclusion (SE) Conflicts

If the exclusion rules were used to resolve a reduce-reduce conflict then the items involved in that resolution are reported before each state preceded by the marker "----". Violations of the tests for simple exclusion grammars are flagged as are conflicts with the marker ">>>> SE". There are three simple exclusion tests applied for each exclusion rule "E # F":

Test 1 – The right part, F, of the exclusion rule must be a single nonterminal. It may not be a string of symbols, or a terminal.

Test 2a – Ensure that no recognition of E starts during a recognition of F. This means that there must by no states in the parser that contain an item of the form [A = alpha . E beta] and either (i) an item of the form [B = gamma X . Y delta] where B is a descendent of F, or (ii) an item of the form [B = gamma .] where B is a descendent of F that can appear in a nonfinal position of a sentential form generated by F.

Test 2b – Ensure that no recognition of F starts during a recognition of E. This test is the same as Test 2a, but exchange E and F.

Test 3 – Ensure that whenever a recognition of F has completed, a concurrent recognition of E also ends. This can be done by ensuring that no descendents of E are in FOLLOW(F) except perhaps E itself.

One of these three test numbers will be given with each reported SE conflict.

F.5.1.2. Simple-Adjacency-Restriction (SAR) Conflicts

Violations of the tests for simple adjacency-restriction grammars are reported with the marker ">>>> SAR". There are three tests applied for each adjacency restriction "W -/- X":

Test 1 – The left part, W, may not be a terminal symbol.

Test 2 – No state may contain a complete item of the form [W = alpha .] with lookahead set L, such that some symbol in L can appear as the first symbol of a sentential form generated by X.

Test 3 – Any state with an item of the form [B = beta .] with lookahead set L where B could appear at the end of a sentential form generated by W, may not also contain an item of the form [A = . alpha] added during noncanonical state expansion, such that A is in L, and either (i) X could appear as the first symbol of a sentential form generated by alpha, or (ii) alpha can generate the empty string.

One of these three test numbers will be given with each reported SAR conflict.

F.5.2. Other Listing Information

After the listing of the states, a listing is given of the lookahead symbols deleted for reduce actions in each state. This kind of table compression can be suppressed with the program option "-u".

Next is the "come-from" table. For each state q, it lists the number of each state with a transition to state q. This table is useful when trying to find the origin of parser conflicts. An unresolvable NSLR conflict in a state q can often be eliminated by correcting an SLR conflict in a state with a transition to q. The come-from table helps locate all such states. The come-from table has the title "Predecessors of each state".

Finally comes a listing of statistics about the parser. Some of these statistics are repeated after each pass of the vertical or horizontal compression algorithm, if those compressions were requested.

F.6. The Parse-Table File

If there are no errors in the grammar, the information necessary to build a parser is placed in the parse-table file. The parse-table file will have same name as the input file with the file type changed to ".tbl". If the input grammar was taken from standard input then the parse-table file will have the name "soap.tbl". This file is intended to be easily machine readable.

The first record in the file contains three integers:

number-of-symbols number-of-grammar-rules number-of-states

Following this line comes a list of the symbols in the grammar, the productions in the grammar, and the states of the parser. Each symbol, rule, or state occupies one line of the file. The symbols, rules,

and states are not explicitly numbered in this file, instead they are implicitly numbered by their ordering, starting with the index zero.

In the list of symbols, each symbol is preceded by the number 1 if it is a nonterminal, and 0 if it is a terminal.

Each rule appears in the following format:

left-part length-of-right-part right-part-symbols

Each state line contains the number of transitions and reductions in the state, and then pairs of numbers representing each transition or reduction. A transition is represented by the pair of numbers:

symbol destination-state

A reduce action is represented by the pair of numbers:

symbol –rule

The rule number is negative to distinguish it from a shift transition.

The information about the semantic actions comes next. This information consists of:

(1) A line containing the number of different semantic action names that appeared in the grammar, followed by a list of those names.

(2) A list of numbers, one for each rule, that gives the number of the semantic action that applies to each rule. The number 0 means the default semantic action.

Finally the lookahead-symbol map generated by horizontal table compression is output. If horizontal compression was not requested, this table is omitted. Otherwise it consists of one number for each symbol in the grammar. The number gives the parse-table column that should be used for each lookahead symbol.

Earlier versions of this parser generator did not allow the specification of semantic actions or horizontal table compression. The information for these two parser features is attached to the end of the parse table in order that the parse tables produced could be accepted by parse-table processors written for the old format.

F.7. Grammar Preparation Tips

Here are some useful tips for writing single-phase grammars for SOAP.

F.7.1. Defining Reserved Words

The following sequence of rules for excluding reserved words from identifiers is **incorrect**.

```
ID = letter | ID letter | ID digit
ID # keyword
```

The reason that this is incorrect is that it would disallow the identifiers that begin with a keyword. The proper definition of identifiers would exclude reserved keywords from completed identifiers only. To do this use rules more like the following:

```
ID = letter | ID letter | ID digit
Finished_ID = ID
Finished_ID # keyword
```

F.7.2. Eliminating NSLR Conflicts

As difficult as eliminating SLR conflicts is for some grammars, eliminating NSLR conflicts can be even harder. A grammar writer should try to eliminate SLR conflicts as much as possible before attacking the NSLR conflicts. Often an unintentional SLR conflict can generate numerous NSLR conflicts that are very hard to resolve. When eliminating an NSLR conflict, check the come-from

table for all the states that have a path to the problem state. Sometimes an SLR conflict in one of these parent states can be much easier to eliminate than the NSLR conflicts that it generates.

F.8. A Sample Grammar and Its Output

F.8.1. The Input File

```
!
!  Sample input file.
!

SOAP 1

S'      = <BOF>  S "\n"  <EOF>      @ exit

S       = TOKEN  |  S TOKEN

TOKEN   = ID white       @ count_id
        | kw1 white      @ count_kw
        | kw2 white      @ count_kw
        | op1 white      @ count_op

letter  = "a"            @ save_char @
        | "b" | "c" | "d"

ID      = id
id      = letter         @ init_id
        | id letter      @ append_letter

kw1     = "bad"
kw2     = "cab"
op1     = "+"

white   =    | " "

! Disambiguation rules.

ID kw1 kw2  -/-  ID kw1 kw2

ID  #  kw1 | kw2
```

F.8.2. The Listing File

```
NSLR(1) parser tables requested.

Symbols:
    0 NT S'
    1  T <BOF>
    2 NT S
    3  T "\n"
    4  T <EOF>
    5 NT TOKEN
    6 NT ID
    7 NT white
```

```
     8 NT kw1
     9 NT kw2
    10 NT op1
    11 NT letter
    12  T "a"
    13  T "b"
    14  T "c"
    15  T "d"
    16 NT id
    17  T "+"
    18  T "\s"

Rules:
    0 S' = <BOF> S "\n" <EOF>     @ exit
    1 S = TOKEN
    2       | S TOKEN
    3 TOKEN = ID white     @ count_id
    4       | kw1 white    @ count_kw
    5       | kw2 white    @ count_kw
    6       | op1 white    @ count_op
    7 letter = "a"     @ save_char
    8       | "b"     @ save_char
    9       | "c"     @ save_char
   10       | "d"     @ save_char
   11 ID = id
   12 ID # kw1
   13 ID # kw2
   14 id = letter     @ init_id
   15       | id letter     @ append_letter
   16 kw1 = "b" "a" "d"
   17 kw2 = "c" "a" "b"
   18 op1 = "+"
   19 white =
   20       | "\s"

No invisible symbols.

Adjacency-restriction rules:
 ID kw1 kw2  -/-  ID kw1 kw2

Exclusion rule:
ID  #  kw1 | kw2

State Graph:

State 0:
        -C- (18,0) op1 = . "+"    --> state 13
        -C- (17,0) kw2 = . "c" "a" "b"    --> state 10
        -C- (16,0) kw1 = . "b" "a" "d"    --> state 9
        -C- (15,0) id = . id letter    --> state 12
        -C- (14,0) id = . letter    --> state 7
        -C- (13,0) ID # . kw2    --> state 5
        -C- (12,0) ID # . kw1    --> state 4
        -C- (11,0) ID = . id    --> state 12
```

```
        -C- (10,0) letter = . "d"    --> state 11
        -C- (9,0) letter = . "c"    --> state 10
        -C- (8,0) letter = . "b"    --> state 9
        -C- (7,0) letter = . "a"    --> state 8
        -C- (6,0) TOKEN = . op1 white    --> state 6
        -C- (5,0) TOKEN = . kw2 white    --> state 5
        -C- (4,0) TOKEN = . kw1 white    --> state 4
        -C- (3,0) TOKEN = . ID white    --> state 3
        -C- (2,0) S = . S TOKEN    --> state 1
        -C- (1,0) S = . TOKEN    --> state 2
        K-- (0,1) S' = <BOF> . S "\n" <EOF>    --> state 1

State 1:
        -C- (18,0) op1 = . "+"    --> state 13
        -C- (17,0) kw2 = . "c" "a" "b"    --> state 10
        -C- (16,0) kw1 = . "b" "a" "d"    --> state 9
        -C- (15,0) id = . id letter    --> state 12
        -C- (14,0) id = . letter    --> state 7
        -C- (13,0) ID # . kw2    --> state 5
        -C- (12,0) ID # . kw1    --> state 4
        -C- (11,0) ID = . id    --> state 12
        -C- (10,0) letter = . "d"    --> state 11
        -C- (9,0) letter = . "c"    --> state 10
        -C- (8,0) letter = . "b"    --> state 9
        -C- (7,0) letter = . "a"    --> state 8
        -C- (6,0) TOKEN = . op1 white    --> state 6
        -C- (5,0) TOKEN = . kw2 white    --> state 5
        -C- (4,0) TOKEN = . kw1 white    --> state 4
        -C- (3,0) TOKEN = . ID white    --> state 3
        K-- (2,1) S = S . TOKEN    --> state 15
        K-- (0,2) S' = <BOF> S . "\n" <EOF>    --> state 14

State 2:
        K-- (1,1) S = TOKEN .    {"\n", TOKEN, ID, kw1, kw2,
              op1, letter, "a", "b", "c", "d", id, "+"}

State 3:
        -C- (20,0) white = . "\s"    --> state 17
        -C- (19,0) white = .    {"\n", TOKEN, ID, kw1, kw2,
              op1, letter, "a", "b", "c", "d", id, "+"}
        K-- (3,1) TOKEN = ID . white    --> state 16

State 4:
        -C- (20,0) white = . "\s"    --> state 17
        -C- (19,0) white = .    {"\n", TOKEN, ID, kw1, kw2,
              op1, letter, "a", "b", "c", "d", id, "+"}
        K-- (12,1) ID # kw1 .    {}
        K-- (4,1) TOKEN = kw1 . white    --> state 18

State 5:
        -C- (20,0) white = . "\s"    --> state 17
        -C- (19,0) white = .    {"\n", TOKEN, ID, kw1, kw2,
              op1, letter, "a", "b", "c", "d", id, "+"}
        K-- (13,1) ID # kw2 .    {}
```

```
        K-- (5,1) TOKEN = kw2 . white    --> state 19

State 6:
        -C- (20,0) white = . "\s"    --> state 17
        -C- (19,0) white = .    {"\n", TOKEN, ID, kw1, kw2,
                op1, letter, "a", "b", "c", "d", id, "+"}
        K-- (6,1) TOKEN = op1 . white    --> state 20

State 7:
        K-- (14,1) id = letter .    {"\n", TOKEN, white,
                op1, letter, "a", "b", "c", "d", "+", "\s"}

State 8:
        K-- (7,1) letter = "a" .    {"\n", TOKEN, white,
                op1, letter, "a", "b", "c", "d", "+", "\s"}

++++ SLR inconsistencies in state 9
        K-- (16,1) kw1 = "b" . "a" "d"
        K-- (8,1) letter = "b" .    {"a"}

State 9:
        K-- (16,1) kw1 = "b" . "a" "d"    --> state 21
        K-- (8,1) letter = "b" .    {"\n", TOKEN, white,
                op1, letter, "b", "c", "d", "+", "\s"}
        --N (7,0) letter = . "a"    --> state 21

++++ SLR inconsistencies in state 10
        K-- (17,1) kw2 = "c" . "a" "b"
        K-- (9,1) letter = "c" .    {"a"}

State 10:
        K-- (17,1) kw2 = "c" . "a" "b"    --> state 22
        K-- (9,1) letter = "c" .    {"\n", TOKEN, white,
                op1, letter, "b", "c", "d", "+", "\s"}
        --N (7,0) letter = . "a"    --> state 22

State 11:
        K-- (10,1) letter = "d" .    {"\n", TOKEN, white,
                op1, letter, "a", "b", "c", "d", "+", "\s"}

State 12:
        K-- (15,1) id = id . letter    --> state 23
        K-- (11,1) ID = id .    {"\n", TOKEN, white, op1,
                "+","\s"}
        -C- (10,0) letter = . "d"    --> state 11
        -C- (9,0) letter = . "c"    --> state 25
        -C- (8,0) letter = . "b"    --> state 24
        -C- (7,0) letter = . "a"    --> state 8

State 13:
        K-- (18,1) op1 = "+" .    {"\n", TOKEN, ID, white,
                kw1, kw2, op1, letter, "a", "b", "c", "d",
                id, "+", "\s"}
```

```
State 14:
        K-- (0,3) S' = <BOF> S "\n" . <EOF>    --> state 26

State 15:
        K-- (2,2) S = S TOKEN .    {"\n", TOKEN, ID, kw1,
                kw2, op1, letter, "a", "b", "c", "d", id,
                "+"}

State 16:
        K-- (3,2) TOKEN = ID white .    {"\n", TOKEN, ID,
                kw1, kw2, op1, letter, "a", "b", "c", "d",
                id, "+"}

State 17:
        K-- (20,1) white = "\s" .    {"\n", TOKEN, ID, kw1,
                kw2, op1, letter, "a", "b", "c", "d", id,
                "+"}

State 18:
        K-- (4,2) TOKEN = kw1 white .    {"\n", TOKEN, ID,
                kw1, kw2, op1, letter, "a", "b", "c", "d",
                id, "+"}

State 19:
        K-- (5,2) TOKEN = kw2 white .    {"\n", TOKEN, ID,
                kw1, kw2, op1, letter, "a", "b", "c", "d",
                id, "+"}

State 20:
        K-- (6,2) TOKEN = op1 white .    {"\n", TOKEN, ID,
                kw1, kw2, op1, letter, "a", "b", "c", "d",
                id, "+"}

++++ SLR inconsistencies in state 21
        K-- (16,2) kw1 = "b" "a" . "d"
        K-- (7,1) letter = "a" .    {"d"}

State 21:
        K-- (16,2) kw1 = "b" "a" . "d"    --> state 27
        --N (10,0) letter = . "d"    --> state 27
        K-- (7,1) letter = "a" .    {"\n", TOKEN, white,
                op1, letter, "a", "b", "c", "+", "\s"}

++++ SLR inconsistencies in state 22
        K-- (17,2) kw2 = "c" "a" . "b"
        K-- (7,1) letter = "a" .    {"b"}

State 22:
        K-- (17,2) kw2 = "c" "a" . "b"    --> state 28
        --N (8,0) letter = . "b"    --> state 28
        K-- (7,1) letter = "a" .    {"\n", TOKEN, white,
                op1, letter, "a", "c", "d", "+", "\s"}

State 23:
```

```
        K-- (15,2) id = id letter .   {"\n", TOKEN, white,
                op1, letter, "a", "b", "c", "d", "+", "\s"}

State 24:
        K-- (8,1) letter = "b" .   {"\n", TOKEN, white,
                op1, letter, "a", "b", "c", "d", "+", "\s"}

State 25:
        K-- (9,1) letter = "c" .   {"\n", TOKEN, white,
                op1, letter, "a", "b", "c", "d", "+", "\s"}

State 26:
        K-- (0,4) S' = <BOF> S "\n" <EOF> .   {}

---- Exclusion of reduce actions in state 27.
        K-- (10,1) letter = "d" .   {"\n", TOKEN, white,
                op1, "+", "\s"}
State 27:
        K-- (16,3) kw1 = "b" "a" "d" .   {"\n", TOKEN,
                white, op1, "+","\s"}
        K-- (10,1) letter = "d" .   {letter, "a", "b", "c",
                "d"}

---- Exclusion of reduce actions in state 28.
        K-- (8,1) letter = "b" .   {"\n", TOKEN, white,
                op1,"+", "\s"}

State 28:
        K-- (17,3) kw2 = "c" "a" "b" .   {"\n", TOKEN,
                white, op1, "+","\s"}
        K-- (8,1) letter = "b" .   {letter, "a", "b", "c",
                "d"}


Lookahead-Symbols Deleted.

State   Symbols
-----   -------

2       TOKEN, ID, kw1, kw2, op1, id
3       TOKEN, ID, kw1, kw2, op1, id
4       TOKEN, ID, kw1, kw2, op1, id
5       TOKEN, ID, kw1, kw2, op1, id
6       TOKEN, ID, kw1, kw2, op1, id
7       TOKEN, white, op1
8       TOKEN, white, op1, letter
9       TOKEN, white, op1
10      TOKEN, white, op1
11      TOKEN, white, op1, letter
12      TOKEN, white, op1
13      TOKEN, ID, white, kw1, kw2, op1, letter, id
15      TOKEN, ID, kw1, kw2, op1, id
16      TOKEN, ID, kw1, kw2, op1, id
17      TOKEN, ID, kw1, kw2, op1, letter, id
```

```
18      TOKEN, ID, kw1, kw2, op1, id
19      TOKEN, ID, kw1, kw2, op1, id
20      TOKEN, ID, kw1, kw2, op1, id
21      TOKEN, white, op1
22      TOKEN, white, op1
23      TOKEN, white, op1
24      TOKEN, white, op1, letter
25      TOKEN, white, op1, letter
27      TOKEN, white, op1, letter
28      TOKEN, white, op1, letter

Predecessors of each state.
0:
1:    0
2:    0
3:    0    1
4:    0    1
5:    0    1
6:    0    1
7:    0    1
8:    0    1   12
9:    0    1
10:    0    1
11:    0    1   12
12:    0    1
13:    0    1
14:    1
15:    1
16:    3
17:    3    4    5    6
18:    4
19:    5
20:    6
21:    9
22:   10
23:   12
24:   12
25:   12
26:   14
27:   21
28:   22


  Processing statistics:
                          Maximum  Average
  Kernel size:            3        1.344828
  SLR closure size:       20       2.931035
  NSLR closure size:      20       3.068965
  State hash-chain len:   2        1.035714


 Conflict Summary
 ----------------
4 SLR inconsistencies.
```

```
Grammar passes SAR test 1.
Grammar passes SAR test 2.
Grammar passes SAR test 3.

Number of parser rows:    29
Number of parser columns: 19
Number of non-ERROR parser actions: 216
      44 shifts actions
      16 reductions on nonterminal lookahead symbols
     156 reductions on terminal lookahead symbols

Vertical table compression -- pass 1.

Number of parser rows:    25
Number of parser columns: 19
Number of non-ERROR parser actions: 188
      44 shifts actions
      15 reductions on nonterminal lookahead symbols
     129 reductions on terminal lookahead symbols

Vertical table compression -- pass 2.

Number of parser rows:    24
Number of parser columns: 19
Number of non-ERROR parser actions: 179
      42 shifts actions
      14 reductions on nonterminal lookahead symbols
     123 reductions on terminal lookahead symbols

Vertical table compression -- pass 3.

Number of parser rows:    24
Number of parser columns: 19
Number of non-ERROR parser actions: 179
      42 shifts actions
      14 reductions on nonterminal lookahead symbols
     123 reductions on terminal lookahead symbols

Horizontal table compression applied:

Number of parser rows:    24
Number of parser columns: 17
Number of non-ERROR parser actions: 177
      40 shifts actions
      14 reductions on nonterminal lookahead symbols
     123 reductions on terminal lookahead symbols
```

F.8.3. The Parse-Table File

```
2
19 21 29
1 S'
0 <BOF>
1 S
```

```
0 "\n"
0 <EOF>
1 TOKEN
1 ID
1 white
1 kw1
1 kw2
1 op1
1 letter
0 "a"
0 "b"
0 "c"
0 "d"
1 id
0 "+"
0 "\s"
0 4 1 2 3 4
2 1 5
2 2 2 5
5 2 6 7
5 2 8 7
5 2 9 7
5 2 10 7
11 1 12
11 1 13
11 1 14
11 1 15
6 1 16
6 1 8
6 1 9
16 1 11
16 2 16 11
8 3 13 12 15
9 3 14 12 13
10 1 17
7 0
7 1 18
13 2 1 5 2 6 3 8 4 9 5 10 6 11 7 12 8 13 9 14 10 15 11 16\
 12 17 13
13 3 14 5 15 6 3 8 4 9 5 10 6 11 7 12 8 13 9 14 10 15 11 16\
 12 17 13
7 3 -1 11 -1 12 -1 13 -1 14 -1 15 -1 17 -1
9 3 -19 7 16 11 -19 12 -19 13 -19 14 -19 15 -19 17 -19 18 17
9 3 -19 7 18 11 -19 12 -19 13 -19 14 -19 15 -19 17 -19 18 17
9 3 -19 7 19 11 -19 12 -19 13 -19 14 -19 15 -19 17 -19 18 17
9 3 -19 7 20 11 -19 12 -19 13 -19 14 -19 15 -19 17 -19 18 17
8 3 -14 11 -14 12 -14 13 -14 14 -14 15 -14 17 -14 18 -14
7 3 -7 12 -7 13 -7 14 -7 15 -7 17 -7 18 -7
8 3 -8 11 -8 12 21 13 -8 14 -8 15 -8 17 -8 18 -8
8 3 -9 11 -9 12 22 13 -9 14 -9 15 -9 17 -9 18 -9
7 3 -10 12 -10 13 -10 14 -10 15 -10 17 -10 18 -10
8 3 -11 11 23 12 8 13 24 14 25 15 11 17 -11 18 -11
7 3 -18 12 -18 13 -18 14 -18 15 -18 17 -18 18 -18
1 4 26
```

```
7 3 -2 11 -2 12 -2 13 -2 14 -2 15 -2 17 -2
7 3 -3 11 -3 12 -3 13 -3 14 -3 15 -3 17 -3
6 3 -20 12 -20 13 -20 14 -20 15 -20 17 -20
7 3 -4 11 -4 12 -4 13 -4 14 -4 15 -4 17 -4
7 3 -5 11 -5 12 -5 13 -5 14 -5 15 -5 17 -5
7 3 -6 11 -6 12 -6 13 -6 14 -6 15 -6 17 -6
8 3 -7 11 -7 12 -7 13 -7 14 -7 15 27 17 -7 18 -7
8 3 -7 11 -7 12 -7 13 28 14 -7 15 -7 17 -7 18 -7
8 3 -15 11 -15 12 -15 13 -15 14 -15 15 -15 17 -15 18 -15
7 3 -8 12 -8 13 -8 14 -8 15 -8 17 -8 18 -8
7 3 -9 12 -9 13 -9 14 -9 15 -9 17 -9 18 -9
0
7 3 -16 12 -10 13 -10 14 -10 15 -10 17 -16 18 -16
7 3 -17 12 -8 13 -8 14 -8 15 -8 17 -17 18 -17
8 Default exit count_id count_kw count_op save_char init_id\
 append_letter
 1 0 0 2 3 3 4 5 5 5 5 0 0 0 6 7 0 0 0 0 0
```

Bibliography

1. Aho, Alfred V. and Ullman, Jeffrey D., *Theory of Parsing, Translation, and Compiling. Vols. I and II.* Prentice-Hall, Englewood Cliffs, N.J. (1972).
2. Aho, A. V. and Johnson, S. C., "LR parsing." *ACM Computing Surveys*, Vol. 6, No. 2, pp. 99-124 (June 1974).
3. Aho, A. V., Johnson, S. C., and Ullman, J. D., "Deterministic parsing of ambiguous grammars." *Communications of the ACM*, Vol. 18, No. 8, pp. 441-452 (Aug. 1975).
4. Aho, Alfred V. and Ullman, Jeffrey D., *Principles of Compiler Design.* Addison-Wesley, Reading, Massachusetts (1977).
5. Aho, Alfred V., Sethi, Ravi., and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Mass. (1986).
6. ANSI, X3.53-1976, *American National Standard Programming Language PL/I.* American National Standard Institute Inc., New York (Aug. 1976).
7. Baker, Theodore P., "Extending lookahead for LR parsers." *Journal of Computer and System Science*, Vol. 22, No. 2, pp. 243-259 (1982).
8. Beatty, John C., "On the relationship between the LL(1) and LR(1) grammars." *JACM*, Vol. 29, No. 4, pp. 1007-1022 (Oct. 1982).
9. Bermudez, Manuel E. and Schimpf, Karl M., "A practical arbitrary look-ahead LR parsing technique." *ACM SIGPLAN Notices*, Vol. 21, No. 7, pp. 136-144 (July 1986).
10. Colmerauer, Alain., "Total precedence relations." *Journal of the ACM*, Vol. 17, No. 1, pp. 14-30 (Jan. 1970).
11. Culik, Karel. and Cohen, Rina., "LR-regular grammars—an extension of LR(k) grammars." *Journal of Computer and System Sciences*, Vol. 7, No. ?, pp. 66-96, Academic Press (1973).
12. Dencker, Peter., Dürre, Karl., and Heuft, Johannes., "Optimization of parser tables for portable compilers." *ACM TOPLAS*, Vol. 6, No. 4, pp. 546-572 (Oct. 1984).
13. DeRemer, Franklin L., "Simple LR(k) grammars." *Communications of the ACM*, Vol. 14, No. 7, pp. 453-460 (July 1971).
14. DeRemer, Franklin L., "Lexical analysis." in *Compiler Construction: An Advanced Course.*, ed. F. L. Bauer and J. Eickel, pp. 105-120, Springer-Verlag, Berlin (1976).
15. DeRemer, Franklin L. and Pennello, Thomas J., "Efficient computation of LALR(1) look-ahead sets." *ACM TOPLAS*, Vol. 4, No. 4, pp. 615-649 (Oct. 1982).
16. DOD, United States Department of Defense , *Reference Manual for the Ada Programming Language.* (1980).
17. Donnelly, Charles. and Stallman, Richard, *Bison Reference Manual.* Free Software Foundation Inc., 675 Mass Ave., Cambridge, MA 02139 (1989).
18. Dueck, Gerald D. P. and Cormack, Gordon V., "Modular Attribute Grammars." Technical Report CS-88-19, University of Waterloo, Waterloo, Canada (May 1988).
19. Earley, Jay., "An efficient context-free parsing algorithm." *CACM*, Vol. 13, No. 2, pp. 94-102 (Feb. 1970).
20. Ginsburg, Seymour, *The Mathematical Theory of Context-Free Languages.* McGraw-Hill, Inc., New York (1966).
21. Groening, Klaus. and Ohsendoth, Christoph, "NEMO: a nicely modified YACC." *ACM SIGPLAN Notices*, Vol. 21, No. 4, pp. 58-69 (Apr. 1986).

22. Harrison, Michael A., *Introduction to Formal Language Theory.* Addison-Wesley, Reading, Mass. (1978).

23. Heering, J., Klint, P., and Rekers, J., "Incremental generation of parsers." *ACM SIGPLAN Notices*, Vol. 24, No. 7, pp. 179-191 (July 1989). Presented at SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon, June 21-23, 1989.

24. Hopcroft, John E. and Ullman, Jeffrey D., *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass. (1979).

25. Ives, Fred., "Unifying view of recent LALR(1) lookahead set algorithms." *ACM SIGPLAN Notices*, Vol. 21, No. 7, pp. 131-135 (July 1986).

26. Jensen, Kathleen and Wirth, Niklaus, *Pascal User Manual and Report. Third Edition.* Springer-Verlag, New York (1985). Revised by: Andrew B. Mickel and James F. Miner

27. Johnson, Stephen C., "Yacc: Yet Another Compiler Compiler." Computer Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey 07974 (1975).

28. Karasick, M. S. and Cormack, G. V., "An efficient parser for reasonably deterministic grammars." *Congressus Numerantum*, Vol. 46, pp. 149-157 (1985).

29. Knuth, Donald E., "On the translation of languages from left to right." *Information and Control*, Vol. 8, No. ?, pp. 607-639 (1965).

30. Koster, C. H. A., "Affix-grammars." in *ALGOL-68 Implementation*, ed. J. E. L. Peck, pp. 95-109, North-Holland, Amsterdam (1971).

31. Koster, C. H. A., "Two-level grammars." in *Compiler Construction: An Advanced Course.*, ed. F. L. Bauer and J. Eickel, pp. 146-156, Springer-Verlag, Berlin (1976).

32. Kristensen, B. B. and Madsen, O. L., "Methods for computing LALR(k) look-ahead." *ACM TOPLAS*, Vol. 3, No. 1, pp. 60-82 (Jan. 1981).

33. Krzemien, Roman. and Lukasiewicz, Andrzej., "Automatic generation of lexical analyzers in a compiler-compiler." *Information Processing Letters*, Vol. 4, No. 6, pp. 165-168 (Mar. 1976).

34. Kusters, Norbert P. and Cormack, Gordon V. , *LA-NSLR.* (May 1988). An LALR(1) parser generator with simple noncanonical conflict resolution.

35. LaLonde, Wilf R., "Regular right part grammars and their parsers." *CACM*, Vol. 20, No. 10, pp. 731-741 (Oct. 1977).

36. LaLonde, Wilf R., "Constructing LR parsers for regular right part grammars." *Acta Informatica*, Vol. 11, pp. 177-193, Springer-Verlag (1979).

37. Lalonde, Wilf R., "The construction of stack-controlling LR parsers for regular right part grammars." *ACM TOPLAS*, Vol. 3, No. 2, pp. 168-206 (Apr. 1981).

38. Lesk, M. E., "LEX—A Lexical Analyzer Generator." Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J. (Oct. 1975).

39. Mavaddat, Farhad., "A unified approach to evaluation of expressions." *BIT*, Vol. 12, pp. 204-212 (1972).

40. Mössenbock, H., "Alex - a simple and efficient scanner generator." *ACM SIGPLAN Notices*, Vol. 21, No. 12, pp. 139-148 (Dec. 1986).

41. Nijholt, A., "On the relationship between LL(k) and LR(k) grammars." *Information Processing Letters*, Vol. 15, No. 3, pp. 97-101 (Oct. 1982).

42. Pager, D., "A practical general method for constructing LR(k) parsers." *Acta Informatica*, Vol. 7, pp. 249-268 (1977).

43. Park, Joseph C. H., Choe, K. M., and Chang, C. H., "A new analysis of LALR formalisms." *ACM TOPLAS*, Vol. 7, No. 1, pp. 159-175 (Jan. 1985).

44. Parnas, D.L., "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, Vol. 15, No. 10, pp. 1053-1058 (1972).

45. Paxson, Vern. , *The FLEX Scanner Generator (Program)*. (May 1987). Real Time Systems, Bldg 46A, Lawrence Berkeley Laboratory, 1 Cyclotron Rd., Berkeley, CA 94720.

46. Pennello, Thomas J. and DeRemer, Frank., "A forward move algorithm for LR error recovery." *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp. 241-254 (1978).

47. Pennello, Thomas J., *In personal communications*. (Mar. 1979).

48. Pennello, Thomas J., "Very fast LR parsing." *ACM SIGPLAN Notices*, Vol. 21, No. 7, pp. 145-151 (July 1986).

49. Salomon, Daniel J. and Cormack, Gordon V., "Scannerless NSLR(1) Parsing of Programming Languages." *ACM SIGPLAN Notices*, Vol. 24, No. 7, pp. 170-178 (July 1989). Presented at SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon, June 21-23, 1989.

50. Salomon, Daniel J. and Cormack, Gordon V., "Corrections to the Paper: Scannerless NSLR(1) Parsing of Programming Languages." *ACM SIGPLAN Notices*, Vol. 24, No. 11, pp. 80-83 (Nov. 1989).

51. Share, Michael., "Resolving ambiguities in the parsing of translation grammars." *ACM SIGPLAN Notices*, Vol. 23, No. 8, pp. 103-109 (Aug. 1988).

52. Spector, David., "Efficient full LR(1) parser generation." *ACM SIGPLAN Notices*, Vol. 23, No. 12, pp. 143-150 (Dec. 1988).

53. Stearns, R. E., "A regularity test for pushdown machines." *Information and Control*, Vol. 11, No. 3, pp. 323-340 (Sep. 1967).

54. Szymanski, Thomas G. and Williams, John H., "Noncanonical extensions of bottom-up parsing techniques." *SIAM Journal of Computing*, Vol. 5, No. 2, pp. 231-250 (June 1976).

55. Tai, Kuo-Chung., "Noncanonical SLR(1) Grammars." *ACM TOPLAS*, Vol. 1, No. 2, pp. 295-320 (Oct. 1979).

56. Valiant, Leslie G., "Regularity and related problems for deterministic pushdown automata." *Journal of the ACM*, Vol. 22, No. 1, pp. 1-10 (Jan. 1975).

57. Wharton, R. M., "Resolution of ambiguity in parsing." *Acta Informatica*, Vol. 6, No. 4, pp. 387-395 (Apr. 1976).

58. Whitney, M. and Horspool, R.N., "Extremely rapid LR parsing." in *Proceedings of Workshop on Compiler-Compiler and High-Speed Compilation.*, Berlin, G.D.R. (Oct. 1988).

59. Wirth, Niklaus, *Programming in Modula-2. Third Corrected Edition.* Springer-Verlag, New York (1985).