OPTIMIZATION OF DIRECTLY EXECUTABLE LR PARSERS

by

Michael James Whitney
B.Sc., University of Alberta, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

ACCEPTED
FACULTY OF GRADUATE STUDIES
[redacted]
DEAN
DATE Sept 13, 1988

in the Department
of
Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. R. Nigel Horspool

[redacted]

(for) Dr. Hausi A. Müller

[redacted]

Dr. Geoffrey W. Vickers

[redacted]

Dr. Panajotis Agathoklis

ISBN 0-315-46502-6

ABSTRACT

Directly-executable parsers enable compilers and other parse-intensive applications to run significantly faster than those which use table-driven parsers. The cost for increased speed is a larger parser. It is possible to further increase parsing speed by applying various optimization algorithms at parser generation time. Some existing optimizations used with table-driven parsers are applicable, while others apply only to directly-executable parsers. It is also possible to reduce the size of hard-coded parsers substantially.

Several optimization algorithms for directly-executable *LR* parsers are presented. They are designed to be implementable at the parser construction phase, within an *LR* parser generator. The algorithms are "adaptive" in the sense that the complexity of each part of the generated parsing automaton reflects the complexity of the grammatical construct to be parsed, independently of the rest of the parser.

Supervisor: **Dr. R.N. Horspool**

Examiners:

[redacted]

Dr. R. Nigel Horspool

[redacted]

Dr. Hausi A. Müller

[redacted]

Dr. Geoffrey W. Vickers

[redacted]

Dr. Panajotis Agathoklis

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENT

I would like to thank the Natural Sciences and Engineering Research Council of Canada for supporting my graduate work.

CHAPTER 1

INTRODUCTION

1.1. The Cost of Generality

Too often in computing, a sledgehammer is used to drive a thumbtack. Programs are simply too large and too general for the tasks to which they are assigned. As a result, performance in terms of speed of execution suffers. This is especially true of parsers, because they are usually mechanically generated from a specification (a grammar, with associated semantic actions) that can be quite complex, but often is not. Even if the specification is simple, the parser is only as efficient as one generated from a more complicated specification, due to the power but inflexibility of the generator software. This thesis provides a different approach to mechanical parser generation – an approach that uses the generality of powerful parser generation methods, but leaves the resulting parser as simple and as fast as possible for its specific application.

1.2. Parsers and Compilers

A parser is any device that recognizes, and imparts structure to, sentences of a language. In computing science, a parser is a program or routine that accepts a stream of input data (a "sentence"), recognizes this data as conforming to a particular language's grammar, and outputs data with which other routines can determine the meaning of the input stream. Thus, it is the job of the parser to recognize the input and its structure, providing a vehicle for translation.

In software development environments, the most important and widespread parser application is in compilation. A compiler translates programs written in a high-level user language into assembly code programs that are machine-executable. The compiler's parser is used to recognize the syntactic structure of the source program to

be compiled, and provide data structures with which other parts of the compiler can determine the meaning of the program. Current compiler design calls for *syntax-driven translation*, in that all computation occurs as procedure calls from the parser [ASU 86]. In this sense, the parser is the heart of the compiler.

A large fraction of computer resources (and programmers' time) is taken up with the compilation and recompilation of users' programs, so programming a compiler for efficiency should receive no less attention than the programming of, say, file editors. Since parsing is central to the operation of a compiler, a considerable amount of effort should be spent in ensuring that its parser is both correct and efficient. There are other applications for parsers that are more parse-intensive than compilation, for example syntax and consistency checkers such as *lint*,[1] but we discuss parsing primarily in the context of compilation.

Figure 1.1 illustrates the central role of the parser in a compiler. The parser calls

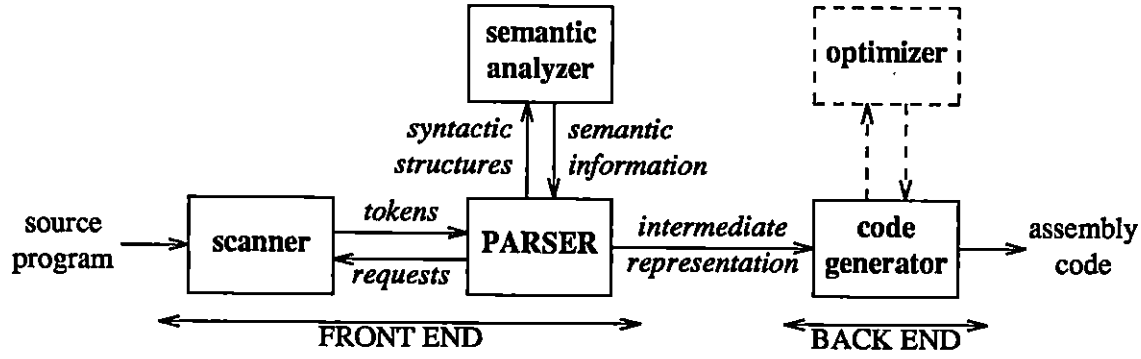


Figure 1.1 : Structure of a Compiler

[1] *Lint* is used to check C programs.

for and receives input from the lexical analyzer, or *scanner*. The job of the scanner, itself a kind of parser, is to group characters in the input stream (a program to be compiled) into entities called *tokens*. Tokens are the "words" of the language that the parser is to recognize. These include keywords (such as `while`), identifiers (such as `line_count`), and special symbols (like " * " and " { "). The scanner parses at the *lexical* level, grouping characters into words, and the parser parses at the *syntactic* level, grouping words into sentences. Scanners and parsers usually comprise separate software modules because the rules and methods for recognizing words are simpler than the rules and methods required for recognizing sentences. Also, separation of these tasks is convenient for a natural decomposition of the compiler's modules [ASU 86].

At well-defined "connection points," the parser calls the semantic analyzer, which is the final stage for deriving the meaning of the source program. Determining these connection points is an important aspect of compiler design, and the design of the parser must permit a high degree of association between syntactic recognition and semantic action [WaG 84]. Semantics of a programming language involve such issues as the meaning of variable names and expressions, and type consistency. The scanner, parser, and semantic analyzer together are often called the *front-end* of a compiler. Data structures augmented with semantic information are output from the parser to the *back-end* of the compiler, which contains the code generator plus possibly an optimizer.

Compiler construction methods have advanced considerably since the introduction of the first compilers in the 1950's. Analytical formalizations have enabled the development of software tools that perform much of the necessary work, especially tools for the automatic generation of scanners and parsers. Input to these software generators are grammatical specifications that define the structure of the language to be recognized. Output is software that actually implements the recognizer, along with a well-defined interface for linking the recognizer to other software components.

Not all parsers for programming languages need be automatically generated. The parsers for some grammars can be written by hand relatively easily. However, these grammars belong to grammar classes that are not sufficiently powerful to describe the syntax of many programming languages. Using a parser generator gives the benefit of superior recognition power, plus it permits easy maintenance and modification of the

generated parser [ASU 86, FiL 88]. Ease of compiler development is increased because actions of the compiler can be based directly on the recognition of grammatical rules. But inefficiency can arise when the parser generator does not have the flexibility to produce simpler parsers for simpler languages. Some extra effort at the generation phase can pay itself off many times over when the generated parsers themselves require fewer computer resources.

Use of software tools for automated software generation is not limited to parsers and scanners alone. For example, code generator generators have been developed to further automate compiler construction. Attribute grammars provide a more unified approach to the syntactic and semantic issues of programming languages, and there is currently a great deal of interest in compiler generation using this method. The Mk* project, currently under development at the University of Victoria, views compiler development as a series of well-defined phases with well-defined interfaces, with each phase independently constructed using a software tool [HoL 87, HoL 88]. Figure 1.2 shows how software tools are used in the development of compilers.

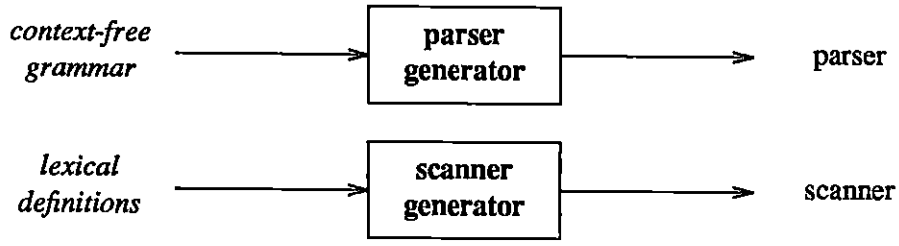


Figure 1.2 : Use of Software Tools for Compiler Development

1.3. The Syntax of Programming Languages

Formal language theory has developed substantially over the last few decades, coinciding with the desire to define programming languages rigorously. With the development of ALGOL in the late '50s and early '60s came the grammar specification known now as *Backus-Naur Form*, or BNF [WiW 66]. BNF specifications describe the *context-free* grammars defined a few years earlier by Chomsky, and are now frequently used to define the syntax of programming languages [ASU 86]. Context-free grammars, in BNF notation or otherwise, are convenient to use as input to parser generators, to create compilers and interpreters of programming languages. The parsing and parser generation techniques described in this thesis are based on context-free grammars.

Formally, a context-free grammar is a four-tuple $< S, V_t, V_n, P >$, where V_t is a finite set of *terminal* symbols, V_n is a finite set of *nonterminal* symbols, S is a unique member of V_n called the *start* symbol, and P is a set of *productions*, or *rules*. Each rule is of the form $A \rightarrow \alpha$, where A is a nonterminal symbol ($A \in V_n$), and α is a string of zero or more grammar symbols ($\alpha \in (V_t \cup V_n)^*$). The *vocabulary* of the context-free grammar is given by $V = V_t \cup V_n$. All symbols occurring on the left-hand-sides of P are nonterminals; the remaining symbols are terminals. Henceforth, the word "grammar" implicitly means "context-free grammar."

Figure 1.3 is an example of a grammar. It describes all arithmetic expressions involving numbers, brackets, and the multiplication and addition operators. Here, the

---

1) $E \rightarrow E + T$
2) $E \rightarrow T$
3) $T \rightarrow T \times F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow num$

Figure 1.3 : Arithmetic Expression Grammar G_1

---

start symbol is E, and P is the set of six rules. By inspection, $V_n = \{E, T, F\}$, and $V_t = \{(,), +, \times, num\}$. Note that there is a many-to-one relationship between grammars and languages, and there are an infinite number of other grammars that generate the same set of expressions. Grammar G_1 is used throughout this thesis to illustrate the effects of optimization techniques.

The symbol "$\Rightarrow$", pronounced "*derives*," is used to generate strings from a grammar. If we have a string $\alpha A \gamma$ and a rule $A \rightarrow \beta$, then $\alpha A \gamma \Rightarrow \alpha \beta \gamma$. A grammar derives strings by beginning with the start symbol, then replacing nonterminals by their right-hand sides, one at a time, until a sentence of the language results. For example, the expression $5 \times (3+4)$ can be derived from grammar G_1 in the following manner:

$$\begin{aligned} E &\Rightarrow T \Rightarrow T \times F \Rightarrow num \times F \Rightarrow num \times (E) \Rightarrow num \times (E+T) \\ &\Rightarrow num \times (E+F) \Rightarrow num \times (T+F) \Rightarrow num \times (F+F) \\ &\Rightarrow num \times (F+num) \Rightarrow num \times (num+num) \end{aligned}$$

At each step of the derivation, an arbitrary nonterminal is chosen for replacement. Each string of symbols along the derivation is called a *sentential form.* The final string is a *sentence*, which is simply a sentential form consisting only of terminal symbols. Obviously, *derives* is a transitive relation. The transitive and reflexive closure of $\Rightarrow$ is denoted $\stackrel{*}{\Rightarrow}$. Now, we can abbreviate the above derivation as

$$E \stackrel{*}{\Rightarrow} num \times (num + num)$$

In this thesis, when discussing grammars, rules, derivations and vocabulary symbols, conventions common to the literature are used:

(1) Nonterminal symbols are represented by upper case letters from the front of the alphabet, *e.g.* A, B, C.

(2) Terminal symbols are represented by lower case letters from the front of the alphabet, *e.g.* a, b, c.

(3) General grammar symbols, both terminal and nonterminal, are represented by upper case letters near the end of the alphabet, *e.g.* X, Y, Z.

(4) Strings consisting of zero or more grammar symbols are represented by lower case Greek letters, *e.g.* α, β, ω, with the exception of ε, which denotes the empty string.

(5) Strings consisting of one or more terminal symbols are represented by lower case letters near the end of the alphabet, *e.g.* w, x, y.

A *parse tree* is often used to show the structure of a sentence. Words of the sentence correspond to leaves of the parse tree, and nonterminal symbols used in the derivation of the sentence become the interior nodes. Given a sentence of a language, and a grammar that generates the language, it is possible to construct a parse tree of that sentence. A parser for a programming language usually outputs data in such a way that the parse tree can be easily reconstructed, or the parser may construct the parse tree directly.

Figure 1.4 shows two possible parse trees for the example expression $5 \times (3+4)$. When deriving the expression, a conventional parser would produce the *concrete* parse tree shown in Figure 1.4 (a). Conceptually, the smaller *abstract syntax* tree of Figure 1.4 (b) is all that is required for translation, and more efficient parsers might generate this type directly. All the required meaning is there in the structure of the tree and the attributes of its nodes, with no superfluous interior nodes.

A parse of a sentence is the inverse operation of a derivation. That is, a parser starts with an input string, and finishes with a parse tree indicating the syntax of that input string. The method used to parse the input string determines the order in which the parse tree is constructed. *Top-down* parsing methods construct the roots first, then the leaves from left to right, *i.e.* in pre-order. This means that derivations are *left-most*: the left-most nonterminal of a sentential form is always generated first [Knu 71]. *Bottom-up* methods recognize the leaves first, then the roots, as in a left first post-order traversal. Derivations are then *right-most* [AhJ 74].

Programming languages are usually designed in such a way as to allow *deterministic* parsing, which means that parsing decisions can always be made based on a finite number of tokens remaining in the input stream. This is not considered a limitation imposed on the designer or user of a programming language, because programs

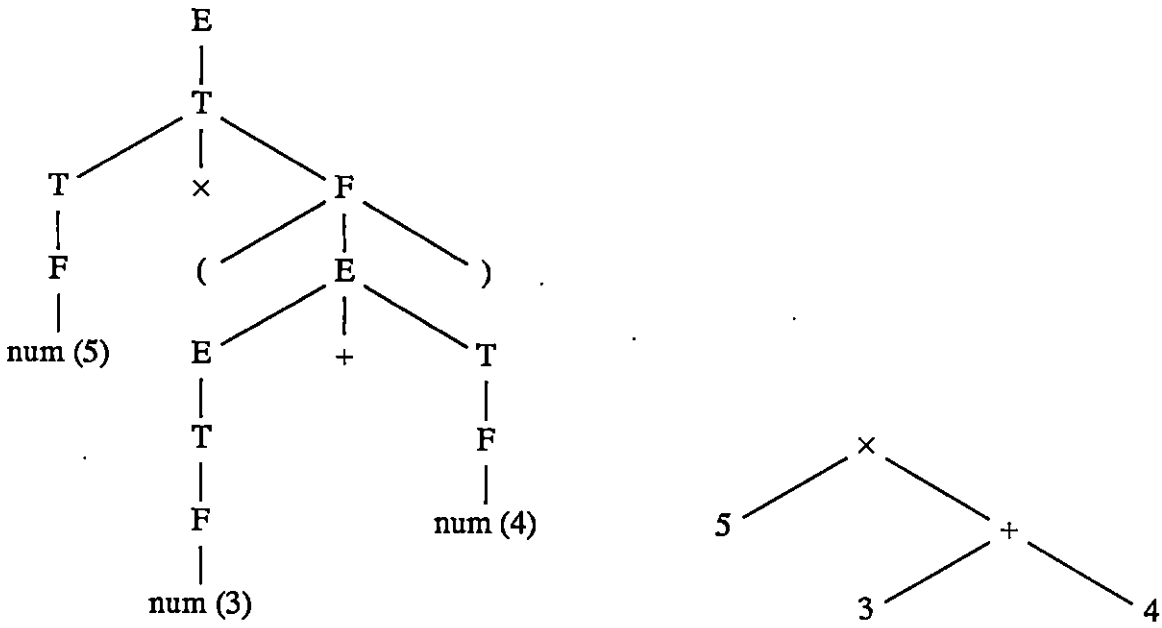


Figure 1.4 : Syntax Trees

that follow simple rules are easier to read, write, and debug, and in general promote more robust software [Wir 85]. A deterministic language can always be recognized by a deterministic finite automaton in conjunction with a push-down stack, together called a *deterministic push-down automaton*, or DPDA [HoU 79]. The stack is required to remember levels of nesting for *self-embedding* derivations (*i.e.*, of the form $A \Rightarrow^{*} \alpha A \omega$, where α and ω are non-empty strings).

The largest known class of languages for which deterministic top-down parsers can be constructed is the *LL(k)* class. Similarly, the *LR(k)* class is the largest known for which deterministic bottom-up parsers can be constructed. Compilers usually employ *LL* or *LR* parsers for the following reasons:

(1) the time required to parse is linearly proportional to the number of tokens in the input stream,

(2) they have the *correct prefix* property, which means that syntax errors are detectable at the first invalid token, and

(3) they can be generated automatically from a context-free grammar, provided that the grammar belongs to the *LL* or *LR* class.

It is possible to hand-code a version of an *LL* parser called *predictive recursive-descent*, but *LR* parsers are practically suited only for mechanical generation. However, *LR* parsers are more powerful in that they are capable of recognizing a much larger class of languages, which includes the *LL* class [AhJ 74, ASU 86].

1.4. The Quest for More Efficient Parsers

In comparison to parsers that have been carefully written by hand, automatically generated parsers suffer from inefficiency in both storage space required and time to execute [WaC 84]. In order to increase speed, or decrease storage space, the compiler implementer is usually faced with the space-time tradeoff dilemma so common in practical computing. There is also a tradeoff between recognition power and efficiency – parsers that recognize a more grammatically complex class of languages are usually larger and slower than those that recognize simpler languages. If various optimization methods are used to reduce or eliminate these disparities, it is important they be implemented automatically by the parser generator. Otherwise, the benefits provided by automatic generation are lost.

Automatically generated parsers (both *LL* and *LR*) are typically *table-driven*. The "state" of the parser is determined by the last table entry that was accessed, and the next state of the parser is determined by the present state and the current grammar symbol. An interpretive loop performs the table look-up operations. The size of the parse tables, and the time required to access them, determine parser efficiency. Optimization methods typically focus on ways to compress the parse tables, since these tables are quite sparse. There is a performance cost for many of these improvements due to increased overhead for table look-up, but such costs are considered well justified due to the enormous amount of space that can be saved [DDH 84].

There exist several methods for increasing the speed of table-driven parsers. First of all, an efficient table look-up interpreter loop is important, because it is executed at least once for every token received from the scanner. For this reason, the loop is sometimes written in assembly code, rather than a higher level programming language. Secondly, fast access methods to the (perhaps compressed) parse tables help. Thus, it is important that table compression techniques take speed of access into account when fast parsing is desired. A third possibility is to eliminate some parser actions altogether, which results in abbreviated output from the parser, for example the abstract syntax tree of Figure 1.4 (b).

Faster parsing times can be realized by abolishing the parse tables entirely. In this case, the state of the parser is implicit in the part of the parser currently being executed, and the next action is determined by comparing the current scan token, or the value at the top of the parse stack, against a usually short list of constants. A parse action takes the form of a simple branch to another part of the parser. Such a parser is called *directly-executable*, or *hard-coded*, reflecting the way in which parsing decisions are made. It can execute more quickly because multiple indexing of arrays to look up parse actions is no longer required [Gra 87].

A hard-coded parser is usually substantially larger than a table-driven parser that recognizes the same language. There are two main reasons:

(1) Since there is no longer an interpretive loop, each parser state requires the code to perform whatever actions are necessary (*e.g.* call the scanner, and/or push a state number onto the parse stack).

(2) Machine instructions that determine actions in a hard-coded parser take up more memory space than table entries. The latter are typically two bytes or less each, whereas the former require a register-constant comparison followed by a conditional branch.

Hard-coded parsers can be easily generated from context-free grammars, using either the *LL* or *LR* techniques. The generator simply outputs code, rather than tables. In fact, it is not difficult to modify the output routines of existing parser generators to make this change.

1.5. Goals of this Research

The goal of this thesis is to prove that directly-executable parsers can be made much faster *and* smaller by applying both established and new optimization techniques at parser generation time. Since we wish to efficiently parse as broad a group of languages as possible, the research is based on the *LR* rather than the *LL* construction. Optimizations are incorporated within the generator so that the methods are applicable to any grammar that is *LR*-parsable, and can be implemented automatically.

The framework of direct execution is chosen for two reasons. First, the main goal is to develop fast parsers, not space-efficient parsers, so a modest increase in size is permissible. The second reason for using direct execution is because with the interpreter loop conspicuously absent in a hard-coded parser, it becomes possible to use optimization methods that are peculiar to certain subsets of states. Most context-free grammars have proper subsets of rules that can be recognized more easily than a general parsing method, applied to the entire grammar, would allow. For example, parsing the regular grammars is quite easy – they are equivalent to deterministic finite automata. Also, there exist the more easily parsed *LL* languages as a subset of the *LR*. As we shall see in Chapters 4 and 5, an *LR* language can be parsed using one of these simpler methods, most of the time. The secondary goal of this thesis is to explore "adaptive" parsing techniques that take advantage of easy-to-parse grammatical constructs whenever they appear.

The optimizations are conveniently split into three main categories. The first is concerned with the improvement of performing parser actions that directly depend upon tokens received from the scanner. The second category deals with reducing use of the parse stack. The last category is concerned with the "streamlining" of parser activity when grammar rules are recognized. All optimizations must be subject to certain constraints so that the usefulness of generated parsers is not compromised. In this thesis, three important rules are adhered to:

(1) The language accepted by the generated parser is precisely that generated by the input grammar.

(2) The ability of *LR* parsers to detect an error as soon as it occurs is not changed.

(3) The parser generator allows semantic action connection points to occur anywhere where a conventional *LR* parser generator would allow.

These three constraints are satisfied in such a way that it need not be visible to a potential user. In other words, the techniques do not require the user to submit any information in addition to the grammar and associated semantics.

Grammar G_1, introduced in Figure 1.3, is used to illustrate many of the techniques developed in this thesis. In addition, the techniques are applied to five more realistic test grammars of programming languages. The largest test grammars are of the C, Pascal, and Oberon languages. There is also a smaller grammar of XPL, and of a subset of Algol.

1.6. Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background material on *LR* parsing, and also discusses techniques used in generating hard-coded *LL* and *LR* parsers. Both assembly language and the higher level language C are considered as target languages. Chapter 3 discusses the use of adapted table-compaction methods for reducing the size of hard-coded parsers, by sharing the code that implements transitions on terminal symbols.

The idea of *minpush* parsing is introduced in Chapter 4. Algorithms for generating minpush parsers, and minpush parsing an input string, are presented. The phenomena of *push-count conflict* rules and states, and their effect on hard-coded parsers, are also presented. Similarities and differences in construction, power and performance, between minpush parsers and other types of parsers, are discussed.

Chapter 5 extends the minpush idea with *direct reduction*, which permits faster recognition of certain rules in context-free grammars. The chapter also presents methods that reduce the amount of time required to make transitions on nonterminal symbols. The results of Chapter 4 are improved using the results of Chapter 5. Finally, Chapter 6 summarizes, and discusses open questions and future work.

CHAPTER 2

BACKGROUND

2.1. LR Parsing

At present, parsers used in compiler development and use are almost always generated using the *LL* or *LR* techniques. A recursive descent version of an *LL* parser can be written by hand [WaC 85], but *LR* parsers are practically suited only for mechanical generation. However, *LR* parsers have several advantages over *LL* parsers. Most importantly, *LR* parsers recognize a larger class of languages. In fact, the *LR* (k) grammars correspond precisely to the deterministic unambiguous context-free grammars [AhJ 74]. Also, context-free grammars require less editing to be acceptable to *LR* rather than *LL* parser generators [FiL 88, LLH 72]. Grammatical ambiguities are easily detected, and the resulting parsers run in linear time, although not quite so fast as *LL* parsers [WaC 84]. Both *LR* and *LL* parsers have the correct prefix property.

The idea of *LR* parsing was first introduced by Knuth [Knu 65], and subsequent simplifications to the original formal theory permitted its use in practical applications [DeR 71, AEH 73]. The *LR* parsing technique is *bottom-up*, which means that the parse tree of a sentence is constructed from the leaves to the root. The order in which the nodes are created corresponds to a post-order traversal of the parse tree. The output of an *LR* parser is this post-order traversal, which also corresponds to a rightmost derivation of the input string [AhJ 74].

At any point in the parse of an input string, an *LR* parser is in a particular *state*. The next action of the parser is determined by the current state and a finite number of tokens remaining in the input, called the *lookahead*. In almost all practical *LR* parsers,

the lookahead is limited to one token, and this is the only case considered here.[1] The next parser action is always one of four possibilities:

(1) **Shift** s: transfer control to state s of the parser. State s is pushed onto the parse stack.

(2) **Reduce** p: report recognition of grammar rule p. The parse stack is popped by the number of grammar symbols on the right-hand side of rule p.

(3) **Accept**: the input string has been successfully recognized. Stop parsing.

(4) **Error**: the current input token is unacceptable. Suspend parsing, and call an error recovery routine.

The actions may be considered as entries in an *LR parse table*, which is indexed by parse state and grammar symbol. Figure 2.1 shows the *LR* parsing algorithm in greater detail. The algorithm is "driven" by the parse table A. Each entry in A has an *action* (SHIFT, REDUCE, ACCEPT, or ERROR), and a *number* (a rule number for REDUCE actions, and state numbers for SHIFT actions). The right-hand side lengths and left-hand side symbols of all grammar rules are contained in the arrays LEN and LHS, respectively. The procedure *scan* () returns the next token in the input stream.

2.2. Construction of LR Parse Tables

In this section we briefly review the construction method for *LR* (0), *SLR* (1), and *LALR* (1) parse tables. These methods are all based on the *LR(0) set of states*, and exactly one token of lookahead. The difference between the three types of parsers comes in the way their tables are constructed. A more complete description of general *LR* (k) parse table construction can be found in most textbooks dealing with compiler construction [Wai 79, ASU 86, FiL 88].

For a given context-free grammar G, parse table construction starts with the generation of all the possible parse states. To this end, it is convenient to define the *augmented grammar* G', created by simply adding the rule $S' \rightarrow \vdash S \dashv$ to G. Here, S' is any nonterminal symbol not occurring elsewhere in G, and $\vdash$ and $\dashv$ are two pseudo-

[1] In general, an *LR* (k) parser bases its parsing decisions on k tokens of lookahead information. In this thesis, *LR* is an abbreviation for *LR* (1).

```
procedure LRparse();
begin

        symbol := scan();
        scan_reqd := TRUE;
        Push 0 onto parse stack; { the first state of the parser }
        a := A[0,symbol];

        repeat
                case a.action of
                SHIFT:          begin
                                        if scan_reqd then
                                                symbol := scan();
                                        scan_reqd := TRUE;
                                        state := a.number;
                                        Push state onto parse stack;
                                        a := A[state,symbol]
                                end;
                REDUCE:         begin
                                        scan_reqd := FALSE;
                                        rule := a.number;
                                        lhs := LHS[rule];
                                        Pop stack by LEN[rule];
                                        state := top of stack;
                                        a := A[state,lhs]
                                end;
                ACCEPT:         return;
                ERROR:          error();
                end { case }
        until FALSE

end; { LRparse }
```

Figure 2.1 : *LR* Parsing Algorithm

terminal symbols, also not occurring elsewhere in G, that indicate the conditions "beginning of input" and "end of input," respectively. The rule $S' \rightarrow \vdash S \dashv$ is considered rule 0, and the rest of the rules in G' are assigned an arbitrary ordering $1, 2, \cdots, p$. We expect the grammar G (and G') to be *reduced*, which means that every rule is involved in the derivation of at least one valid sentence. For example, the grammar

$$S' \rightarrow \vdash S \dashv$$
$$S \rightarrow S\ a \mid S\ b$$

is not reduced, since it cannot generate any sentences (although it can generate sentential forms).

The construction of an *LR* (0) set of states is based on the notion of partially completed rule reductions.

Definition: An *LR(0) item* is a pair $< p, n >$ where p is a production rule, and n is a *position* in this rule. An item is usually written as the production rule with a marker, or *dot*, somewhere on the right-hand side. The item is enclosed in square brackets to make clear the distinction between items and rules. If a rule has m symbols on its right-hand side, then the rule has $m+1$ associated *LR* (0) items. For example, the item $< 0, 1 >$ for an augmented grammar is written $[S' \rightarrow \vdash \bullet S \dashv]$. The sole item of a null production $A \rightarrow \varepsilon$ is written $[A \rightarrow \bullet]$.

Intuitively, the meaning of an item $[A \rightarrow \alpha \bullet \beta]$ is that the right-hand side of the rule has been partially recognized as far as the dot. The *LR* (0) construction algorithm makes a complete set of all possible collections of items that a parse could be in the middle of, starting with the item $[S' \rightarrow \vdash \bullet S \dashv]$. The set of items containing this item is called the *initial state*. Two functions, *closure* and *goto*, are required to generate the *LR* (0) sets of items. (These functions are discussed further in Chapter 4.) Items that have the "dot" anywhere but the beginning of the right-hand side are called *kernel items*, and are generated by the *goto* function applied to item sets. Items that have the "dot" at the beginning of their right-hand side are *closure* items. They are generated by the *closure* function applied to kernel items. Those items that have the "dot" at the

end of their right-hand side are called *reduce* items. The complete set of *LR* (0) item sets generated for the arithmetic expression grammar G_1 is shown in Figure 2.2.

A set of items, such as any of those shown in Figure 2.2, represents an *LR parse state*. It is necessary to convert the set of states to a usable parse table, where rows of the parse table correspond to parse states, columns correspond to grammar symbols, and entries are parse actions. For most item sets, determining the appropriate actions is straightforward. For example, in state (item set) 1 of Figure 2.2, the only valid actions are "shift to state 3 on +," and "shift to state 2 on ⊣." Determining parse actions is more involved for states like state 4, where there is a rule reduction possible because one of the items is a reduce item, but the state also contains at least one other item. State 4 is called *inadequate*, and it is the job of the *SLR* (1) or *LALR* (1) algorithm to prove that the rule $E \rightarrow E + T$ cannot possibly be reduced if the lookahead is the token $\times$, since a shift to state 5 is also possible.

I_0: $[S \rightarrow \vdash \bullet E \dashv]$
$[E \rightarrow \bullet E + T]$
$[E \rightarrow \bullet T]$
$[T \rightarrow \bullet T \times F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet num]$

I_1: $[S \rightarrow \vdash E \bullet \dashv]$
$[E \rightarrow E \bullet + T]$

I_2: $[S \rightarrow \vdash E \dashv \bullet]$

I_3: $[E \rightarrow E + \bullet T]$
$[T \rightarrow \bullet T \times F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$

I_4: $[E \rightarrow E + T \bullet]$
$[T \rightarrow T \bullet \times F]$

I_5: $[T \rightarrow T \times \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet num]$

I_6: $[T \rightarrow T \times F \bullet]$

I_7: $[F \rightarrow (\bullet E)]$
$[E \rightarrow \bullet E + T]$
$[E \rightarrow \bullet T]$
$[T \rightarrow \bullet T \times F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet num]$

I_8: $[F \rightarrow (E \bullet)]$
$[E \rightarrow E \bullet + T]$

I_9: $[F \rightarrow (E) \bullet]$

I_{10}: $[E \rightarrow T \bullet]$
$[T \rightarrow T \bullet \times F]$

I_{11}: $[F \rightarrow num \bullet]$

I_{12}: $[T \rightarrow F \bullet]$

Figure 2.2 : *LR(0)* Sets of Items for Grammar G_1

A grammar that generates a set of states that includes one or more inadequate states is not *LR* (0)-parsable. The conflict that causes an inadequate state can be either a *shift/reduce* conflict, as in the example of state 4 in Figure 2.2, or a *reduce/reduce* conflict, where there exists more than one reduce item in a parse state. The *SLR* (1) (*simple LR(1)*) algorithm generates lookahead sets for rules by grammar analysis [DeR 71]. The *LALR* (1) (*lookahead LR(1)*) algorithm, which is more powerful, generates lookahead sets for reduce items by analysis of the *LR* (0) set of states [AEH 73, AhJ 74]. If *SLR* (1) (or *LALR* (1)) lookahead sets are disjoint for potential conflicts in inadequate *LR* (0) states, a deterministic parser results, and the underlying grammar is said to belong to the *SLR* (1) (or *LALR* (1)) class. The set of *LALR* (1) grammars include the *SLR* (1) grammars, which in turn include the *LR* (0) grammars. Both the *SLR* (1) and *LALR* (1) algorithms result in the table of Figure 2.3 for grammar G_1. Blank entries indicate errors, state 2 is an accept state, and the remaining entries are either shift (S) or reduce (R) actions.

LR (1) parse tables are too large and much more powerful than necessary for driving parsers of practical languages. Thus, variations based on the *LR* (0) set of states are almost always used. However, these parse tables still take up considerably more room than, for example, *LL* (1) tables for a similar grammar. Luckily there exist several optimizations to compress the tables, and improve the operation of the parser, so that the disparity in efficiency is diminished.

Perhaps the most obvious improvement is to break the parse table up into pieces, using matrix factoring and compact representation techniques. Usually, the table A referenced in Figure 2.1 is split into two tables: the *action* table, indexed by terminal symbol and state number, and the *goto* table, indexed by nonterminal symbol and state number. Then, the two tables can usually be compressed to a small fraction of their former size [DDH 84, Jol 74].

Another improvement almost always employed in practical systems is to reduce the number of states in the parser by eliminating *LR(0) reduce-only* states. The basic idea is to bypass transitions into those states where only one valid action is possible – a reduction by exactly one grammar rule. Shift transitions into these states become combined *shift-reduce* transitions. For example, consider again the arithmetic expression

	terminals						nonterminals		
state	*num*	(	)	+	×	⊣	*E*	*T*	*F*
0	S11	S7					S2	S10	S12
1						S3		S3	
2	*accept*								
3		S7						S4	S12
4			R1	R1	S4	R1			
5	S11	S7							S6
6			R3	R3	R3	R3			
7	S11	S7					S8	S10	S12
8			S9	S3					
9			R5	R5	R5	R5			
10			R2	R2	S5	R2			
11			R6	R6	R6	R6			
12			R4	R4	R4	R4			

Figure 2.3 : *SLR(1)* Parse Table for G_1

grammar G_1. The *LR* (0) set of states for this grammar was shown in Figure 2.2. States 6, 9, 11, and 12 are all *LR* (0) reduce-only states, and can therefore be eliminated, and the relevant shift transitions changed, resulting in the *reduced set of states* shown in Figure 2.4 (a). Elimination of reduce-only states is actually part of a broader idea called the *defaulting assumption.* For every row of the action table, the most common parse action can be factored out and used as a default action. For most states, the default is "error," but if one or more of a state's actions calls for a rule reduction, the most common reduction is used as a default action.

A convenient model for an *LR* parser is a graph, where the vertices of the graph are the *LR* states of the parser, and edges between vertices are all the shift transitions, on both terminal and nonterminal symbols, between states. This graph is sometimes called the *goto* graph [ASU 86], and it has also been called the *characteristic finite state*

I_0: $[S \rightarrow \vdash \bullet E \dashv]$
$[E \rightarrow \bullet E + T]$
$[E \rightarrow \bullet T]$
$[T \rightarrow \bullet T \times F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet num]$

I_1: $[S \rightarrow \vdash E \bullet \dashv]$
$[E \rightarrow E \bullet + T]$

I_2: $[E \rightarrow E + \bullet T]$
$[T \rightarrow \bullet T \times F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$

I_3: $[E \rightarrow E + T \bullet]$
$[T \rightarrow T \bullet \times F]$

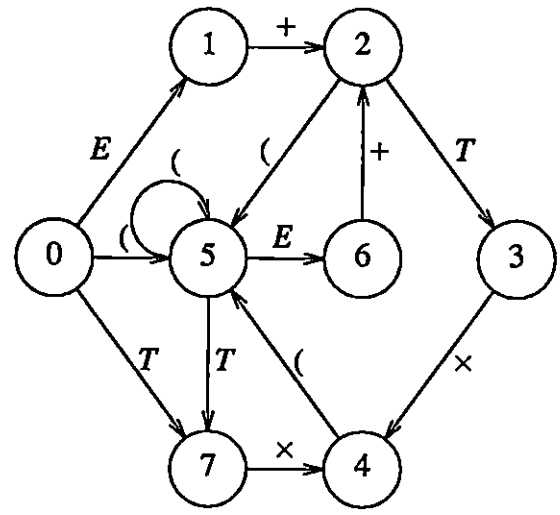
I_4: $[T \rightarrow T \times \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet num]$

I_5: $[F \rightarrow (\bullet E)]$
$[E \rightarrow \bullet E + T]$
$[E \rightarrow \bullet T]$
$[T \rightarrow \bullet T \times F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet num]$

I_6: $[F \rightarrow (E \bullet)]$
$[E \rightarrow E \bullet + T]$

I_7: $[E \rightarrow T \bullet]$
$[T \rightarrow T \bullet \times F]$

(a) Sets of *LR(0)* items



(b) Transition graph

Figure 2.4 : Reduced Parsing Machine

machine, or CFSM [FiL 88]. In this thesis these graphs are called *transition* graphs, to avoid confusion with the *goto* function and table. Transition graphs are really just pictorial representations of deterministic finite automata. The next state of the parser is not always determinable just by transition graph edges, because of rule reduction. However, the next state is always determinable when the graph is considered in conjunction with a parse stack. Together, a DFA and a stack are called a *deterministic push-down automaton*, or DPDA [HoU 79]. The transition graph for grammar G_1 is shown in Figure 2.4 (b).

2.3. Directly Executable Parsers

Both *LR* and *LL* parsers are usually implemented as tables and table-interpreters. Recently there has been some effort to abandon this method in favour of larger, faster parsers that are *directly-executable*. Instead of accessing various tables to determine transitions and subsequent states, a directly-executable parser determines transitions from compare instructions embedded in the code, and transitions are accomplished by simply branching to another part of the parser. Scanners are frequently implemented in this way. The "state" of a directly-executable parser is implicit in the part of the parser currently being executed, and is accessible via labels and branches to this label. For this reason, directly-executable parsers are also called *hard-coded*. In this section we examine techniques that have been used in the mechanical generation of hard-coded parsers of *LL* and *LR* grammars.

Pennello discusses the generation of *LR* parsers, written in assembly language, with the rows of the action and goto tables written in-line as sequences of compare and conditional branch assembly instructions [Pen 86]. The parsers can conceptually be broken down into several components: the *action* blocks, the *goto* blocks, and the *reduce* blocks. Each state that has terminal shift transitions has a corresponding action block. Similarly, each state that has nonterminal shift transitions has a goto block. Grammar rules are recognized in the reduce blocks.

Action blocks consist primarily of sequences of compare and conditional branch instruction pairs. For example, if state 88 has two valid terminal shift transitions on symbols 23 and 46, its action block would look like

```
Q88:  call Scan          ; accessing symbol is a terminal
      push @NTXQ88       ; push address of corresponding goto block
      cmp LA,23
      je Q91             ; destination on 23
      cmp LA,46
      je Q8              ; destination on 46
      jmp ERROR          ; default action
```

Here, we follow Pennello's example by using a generic assembly language, where "**cmp**" compares its two arguments to set the condition codes, "**je**" means "jump if equal," and "**jmp**" is an unconditional branch. Q88, NTXQ88, Q91, Q8 and ERROR

are all symbolic labels corresponding to absolute code addresses occurring elsewhere in the parser. LA is a register that contains the current lookahead token, which is obtained by procedure calls to the "Scan" routine. If the accessing symbol of state 88 is a nonterminal, the "Scan" call is not required. The second instruction pushes the address of the corresponding goto block onto the parse stack.

The last instruction occurring in the action block is always a default action. For state 88, this default is "error," so obviously 88 is an adequate state. Inadequate states use a rule reduction as the default action. Occasionally, inadequate states are *multiply-inconsistent* [Pen 86], which means that there is more than one possible reduction for the state (*i.e.*, the state has a reduce/reduce conflict). If no shift actions are applicable, a search strategy similar to that described below for goto blocks is used to find the default reduction.

Nonterminal transitions are handled a little differently from terminal transitions. Each goto block has an address that is stacked upon entry to the corresponding action block. The address is accessed from the parse stack when a rule reduction is performed. Since not all states have nonterminal transitions, the goto blocks are sometimes empty. For the nonempty goto blocks, a search strategy similar to binary search is used to find the correct nonterminal transition. This is possible because of the well-known result that at least one nonterminal transition must apply; no error is possible [AhJ 74]. Suppose that state 65 has five possible nonterminal transitions on symbols 3, 4, 6, 14, and 20. Then, the goto block for state 65 would look like

```
NTXQ65:
        cmp LHS,4
        je Q56          ; destination on 4
        jl Q21          ; destination on 3
        cmp LHS,14
        je Q78          ; destination on 14
        jl Q87          ; destination on 6
        jmp Q91         ; destination on 20
```

The letters NTX in the block's label stand for "nonterminal transitions." Note that a maximum of two comparisons is required for five nonterminals. In general, the search tree corresponding to the sequence of compare and branch instructions is made as

shallow as possible by using three-way branches. Using this technique, the maximum number of **cmp** instructions required to make a nonterminal transition in a state with n nonterminal transitions is $\lfloor(\log_2 n)\rfloor$, and the expected number of **cmp** instructions is fractionally less.

A binary decision tree is not very useful for terminal transitions in action blocks because the default action must be included once for every leaf of the tree [Pen 86]. There may be so many terminal transitions that the expected execution time for a terminal transition exceeds the time that would be required by an interpretive *LR* parser. In this case, a table look-up scheme is used. The required data structures are 1) an array in which the terminal transitions of all those states with many terminal transitions are "folded" together using table compaction methods, and 2) an array that contains the accessing symbols of these states. A terminal transition then involves indexing into both structures. The technique is identical to that described by Dencker *et al.* as the *row displacement scheme*, or RDS [DDH 84]. Pennello reports that it is useful only when a state has at least nine terminal transitions, because of the overhead for accessing the arrays.

A similar indexing method is used for those goto blocks with too many nonterminal transitions. In terms of space, this does not work out as well as for the terminal transitions, since rows of the goto table do not pack together efficiently [AhJ 74]. However, no default checking is required, so the access is very fast. Since the binary search strategy explained above is also very fast, and because of the space problem, the indexing strategy is used only when a goto block has more than twenty-four nonterminal transitions.

Reduction of grammar rules is handled by the reduce blocks. In Pennello's parsers, the reduce blocks correspond to the first p parse states, where p is the number of production rules. These states correspond to the final states of each rule, and are numbered from 1 up to p. Reduce blocks are accessible via two labels. Suppose rule 2 has nonterminal 15 as its left-hand side, three grammar symbols on its right-hand side, and some terminal symbol as the third symbol on its right-hand side. Then, the reduce block for rule 2 would look as follows:

```
Q2:    call Scan       ; accessed by a terminal
       mov LHS,15      ; load LHS with 15
       jmp R2          ; complete the reduction
NSQ2:  mov LHS,15
       jmp R3
```

Entries into the reduce block at label Q2 correspond to a combined shift-reduction of rule 2. Thus, the stack must be popped by one less than the number of symbols on the right-hand side of rule 2. Entries via the label NSQ2 come from inadequate states where rule 2 is reduced by default, or from multiply-inconsistent states where the reduction of rule 2 is one of a number of possible reductions (NS stands for "no shift"). Like the action blocks, the "Scan" call at label Q2 is not required if the accessing symbol is a nonterminal.

Several additional blocks of code, labeled R0, R1, R2, *etc.*, are required to complete rule reductions. Their only purpose is to pop the stack and perform the indirect branch to a goto block. For example, the block of code referenced by the reduce block above would look like

```
R3:    pop 3
       jmp @(sp)
```

This design saves a minor amount of space compared to simply including the popping code in the reduce blocks. However, it is not clear where code implementing the semantic action associated with a reduction would be placed. The scheme apparently does not allow semantic code (or call to a semantic routine) for a given rule to occur only once in the parser. Any reduce block enterable via both of its labels will have to include semantic code twice. Of course, this problem is eliminated if NSQ labels are used exclusively (*i.e.*, shift-reduce actions are not used), but then the parser will run slightly slower due to some redundant pushing and popping.

Almost all *LR* syntax error recovery methods must have access to the parse stack. The fact that code addresses instead of states are pushed onto the stack complicates error recovery somewhat. Pennello's solution is to include a translation table with the hard-coded parser. The table is always constructable because there is a "Q" address associated with every *LR* parse state. There is considerable overhead involved in

reconstructing the normal parse stack, but the decreased speed is justified due to the supposed infrequency of syntax errors. For those recovery schemes that carry out a number of parallel parses after a syntax error, the standard *LR* parse tables must also be included with the hard-coded parser.

Pennello includes a section on the problem of checking for stack overflow. Since every action block contains a **push** instruction, and these states are entered frequently during the course of a parse, it is expensive to include overflow checking code with each of these instructions. Pennello uses an heuristic algorithm to determine a subset of the action blocks that require overflow checks. There are a number of problems with the method, but we defer discussion of this until Chapter 4.

A different approach to generating hard-coded parsers is taken by Gray [Gra 87]. The generated parsers are *LL(1)*, and they are written in the C programming language instead of assembly code. Since the use of C is so widespread, this means that these parsers have a high degree of portability. However, this portability is compromised through the use of machine-dependent macros. Gray's main objective is to generate fast parsers that incorporate robust syntax error recovery methods. The recovery algorithm used is based on a method defined by Röhrich [Röh 80], which is useful for both *LL* and *LR* parsers.

In a sense, a hard-coded *LL* parser is simply a recursive-descent parser that uses an explicit stack instead of procedure call and return. There is a complication when writing hard-coded stacking parsers in C, because C does not explicitly allow indirect branching. Thus, it is not possible to push return addresses onto the stack in the same way as Pennello does with the NTXQ labels, described previously.[2] Gray uses the "switch" statement in C to solve this problem. The statement allows a program to redirect control to a finite number of "case" labels, depending on the value of the switch parameter. The behaviour mimics indirect branching, although in a somewhat clumsy manner. Gray claims that his parsers run faster than recursive-descent parsers,

[2] This is an inconsistency in the C language. C provides indirect access to function addresses, but not label addresses.

so apparently the scheme is efficient.

"Indirect" indirect branching using a switch statement is less efficient than pure indirect branching for two reasons. First of all, there is the extra overhead of branching to the head of the switch, indexing into the jump table, then finally branching to the desired location. The second problem is that the C compiler must include bounds checking code at the beginning of the jump table, when the parser is compiled. This is because of the indirect reference to the contents of the stack, via the stack pointer. Even a good optimizing compiler cannot do a range analysis on the possible contents of the stack, and thus the bounds checking is required to ensure that program control does not move to a completely unexpected place. In fact, the definition of C states that control must pass exactly to the end of the statement, if no case label is applicable [KeR 78, Hor 86].

2.4. Practical Portable Directly Executable LR Parsers

In this section, we develop the preliminary design of mechanically generated hard-coded *LR* parsers written in C, borrowing from the ideas of both Pennello [Pen 86] and Gray [Gra 87]. The results presented in subsequent chapters are specifically intended for application to parsers of this type. Unlike the approach taken by Gray, we confine ourselves to pure C, without the use of machine-dependent macros. With careful planning, it is possible to "force" a good optimizing C compiler to generate efficient object code.

It is convenient to divide the parser into three main sections: the *actions* part, the *reductions* part, and the *goto* part. Naturally enough, each part consists of the near-equivalent of action blocks, reduce blocks, and goto blocks, respectively, as defined in Section 2.3. The reason we don't follow Pennello's design in coding action blocks and goto blocks in pairs is because all the C goto blocks must be contained within a large switch statement, similar to Gray's design for *LL* parsers. The action blocks and reduce blocks do not need to be within this switch statement, so they are considered separately. Three register variables are required by the parser: `tok`, to hold the lookahead terminal, `lhs`, which holds a nonterminal after a rule is reduced, and `sp`, which is a pointer into the parse stack.

The action part consists of all terminal shift and shift-reduce actions in an *LR* parse table. A symbolic label of the form `st`*n* is used to address the action block of each state, where *n* is the state number. In the previous section, we discussed a hypothetical state 88 in a Pennello parser. The same state in our C parser would look like

```
st88: *++sp = 88;
      tok = scan();
      if (tok==23) goto st91;
      if (tok==46) goto sr8;
      goto error;
```

There are just a few differences between this state 88 and the assembly language state 88. The call to the scanner is slightly more involved, because here the result is explicitly loaded into `tok`. This is necessary to ensure that the C compiler leaves `tok` in a register for fast subsequent register-constant comparisons. The second instruction increments the stack pointer, then stores the state number, instead of the goto block address, at the top of the stack. We would expect that the compare-and-branch `if` statements generate the exact same assembly code that would be in a Pennello parser.

The goto blocks for all those states that have nonterminal transitions are organized within a large "switch" statement, enclosing the entire goto section. The example goto block from Pennello's parser would then look like

```
sw:   switch( *sp ) {
      case 0:    ...
      case 1:    ...
        ...
```

```
case 65:
     if (lhs==4) goto st56;
     if (lhs<4) goto r21;
     if (lhs==14) goto st78;
     if (lhs<14) goto st78;
     goto sr91;
  ...

}
```

Here, the nonterminal comparisons are coded in such a way as to help the C compiler generate efficient code. Hopefully, only one assembly language compare instruction is generated for each pair of C compare-and-branch instructions. If the compiler is smart, assembly language code identical to that given in the previous section for goto block 65 is generated. In order to ensure that the complier generates a jump table for the switch statement, the states are renumbered so that all those states that have one or more non-terminal transitions come first. Thus, all the case labels in the switch statement are numbered consecutively, starting from 0.

The code to reduce rules is organized in quite a different way than in [Pen 86]. Reduce blocks for each rule are put all together in the reduce section. A reduce block can have one or two entry labels, depending on whether a rule can be reduced via a shift-reduce transition only, from an inadequate state only, or both ways. Here is the code for reducing rule 2.

```
sr2:  tok = scan();
      sp++;
r2:   /* semantic code for rule 2 */
      lhs = 15;
      sp -= 3;
      goto sw;
```

Again, the `scan()` call is required because the reduce state for rule 2 is entered via a terminal symbol. Unlike Pennello's design, semantic code need occur only once in this scheme, and the code required to load the `lhs` register and decrement the stack pointer also occurs only once. The only problem with this scheme is the redundant push occurring between the labels. We need not concern ourselves with this problem at present,

because Chapter 4 shows that the instruction can always be eliminated. The indirect branch required to restart the parse is accomplished by branching to the head of the switch statement at label `sw`, where flow of control is directed to the appropriate case label to find the correct nonterminal transition.

Pennello deals with states that have large numbers of transitions by returning to a table-lookup method. It is more involved to do this in C because of the indirect branching problem. It is possible to use one or more switch statements that are assembled into jump tables, but it is complicated to reach an agreeable balance between space used and increased speed. Space optimization techniques applicable to terminal transitions are discussed in Chapter 3. Further discussion of stack use, including the elimination of redundant pushes and checking the stack for overflow, is deferred until Chapter 4.

A C hard-coded *LR* parser is superior to an assembly language *LR* parser for interfacing with a syntax error recovery method. The stack consists of the expected state numbers, not goto block addresses. Note that the default action for the action block of state 88, above, is an unconditional branch to the `error` label, rather than a procedure call to an error recovery routine, as used by Gray [Gra 87]. This saves a significant amount of space, since the error default occurs for most states. Rule reduction defaults are comparatively rare, and procedure calls generate more code than unconditional branches. However, the number of states in which the parser can be restarted is limited to those accessible by branches from a `case` label within the goto part. An alternative design strategy for the *LR* parser written in C is to include the code for *all* states within the switch statement, thus ensuring that restart is possible anywhere. Since we are not concerning ourselves with error recovery issues, the design as described is assumed.

CHAPTER 3

COMPRESSION OF LR ACTION BLOCKS

3.1. Introduction

The chief argument against the use of hard-coded parsers (and thus generators of these parsers) is that they occupy substantially more memory than table-driven ones. Although this might not be a significant drawback for larger machines with vast amounts of random access memory, the argument is relevant if the computer on which the parser is to be run is quite small. Luckily, it is possible to reduce the size of hard-coded parsers substantially, using techniques that are similar to techniques used in the compression of parse action tables. But reduction in size must not come at any significant cost of parsing speed, or the benefit of using the hard-coded parser is compromised. In the case of table-driven parsers, compression methods tend to slow parsing speed due to increased overhead for array indexing. However, the method of *state subsumption*, defined below, has negligible effect on the speed of hard-coded parsers.

3.2. Subsumable States

When compressing *LR* parse tables, it is desirable to use data structures that allow one entry of the table to be accessed by more than one state. The schemes discussed in Chapter 2 achieve this by using various sparse matrix representations, including the overlaying of "compatible" rows, and sometimes columns, and by folding some or all rows into single arrays [DDH 84]. In order to do this, compatibility relations between states are defined. The nature of the compatibility relation depends on how the compressed table is to be structured and accessed.

A hard-coded parser differs from most table-driven ones in that actions are "looked up" via linear search (or possibly binary search) rather than one or more index operations. In order to share hard-coded "actions" (which are simply compare-and-branch instructions), a different kind of compatibility relation must be defined. This relation must take into account the fact that the compare-and-branch instructions that comprise most of the body of the parser each contain two constants: the number of the current grammar symbol, and the label (*i.e.*, the address) of the destination state.[1] Also, the default action must be considered, and finally the consecutive action of the "fetch and execute" architecture that almost all computers employ.

Definition: An *LR* parse state A is said to *subsume* another parse state B *iff* every shift, shift-reduce, and non-default reduce transition (action) in B is also in A, and states A and B have the same default action. Two *LR* parse states A and B are *equal iff* A subsumes B and B subsumes A.

Here, we assume that reduce and shift-reduce transitions are ordered pairs *<symbol, rule>* and shift transitions are ordered pairs *<symbol, state>*. The *subsume* relation is transitive; if A subsumes B and B subsumes C, then A subsumes C. The relation does not take into account any ordering of the transitions contained in either state. Indeed, if ordering was important, the relation would not be nearly as strong or general. Ordering need not be considered because the parser generator can output compare-and-branch instructions in any order that may be convenient, as long as all instructions for a state occur consecutively, no instruction is missing, there are no extra actions, and the default branch instruction comes last.

Figure 3.1 shows how the instructions implementing the transitions of two *LR* parse states in a hard-coded parser are shared when the actions of one state are subsumed by another. Here, state 14 subsumes state 23. Therefore the actions of state 23 can appear within those of state 14, after reordering. We introduce a new class of code labels "`ac..`" to specify the "actions" part of a state's code. This part contains all the compare-and-branch instructions implementing the terminal transitions of a state,

[1] Self-modifying programs are not considered. They defeat the idea of portability.

```
st14: push(14);
      tok = scan();
      if (tok== 2) goto st21;
      if (tok== 3) goto st10;
      if (tok== 7) goto st11;
      if (tok==10) goto sr3;
      if (tok==18) goto st3;
      goto error;
```

```
st23: push(23);
      tok = scan();
      if (tok== 2) goto st21;
      if (tok== 7) goto st11;
      if (tok==10) goto sr3;
      goto error;
```

(a) States 14 and 23 in original hard-coded parser

```
st14: push(14);
      tok = scan();
      if (tok== 3) goto st10;
      if (tok==18) goto st3;
ac23: if (tok== 2) goto st21;
      if (tok== 7) goto st11;
      if (tok==10) goto sr3;
      goto error;
```

```
st23: push(23);
      tok = scan();
      goto ac23;
```

(b) Actions of state 23 subsumed in actions of state 14

Figure 3.1 : Subsumed States

followed by a default action. An incidental cost to parsing speed is incurred by the additional overhead of one unconditional branch in state 23 (*i.e.* `goto ac23`). Note that any state that is equal to state 23 can branch to label `ac23` to find the correct transition.

Subsuming states in the manner illustrated by Figure 3.1 eliminates the possibility of using binary search for most states, because the ordering of compare-and-branch instructions is constrained to suit subsumption. For example, in Figure 3.1 (b) the two actions

```
if (tok== 3) goto st10;
if (tok==18) goto st3;
```

must occur before the `ac23` label. However, the number of states for which binary search is beneficial is usually low. Binary search through lists is only worth implementing when the number of items in the list is quite high, *e.g.* from around ten to sixteen [DDH 84, Pen 86]. Also, using binary search to find terminal transitions uses extra code space, as shown in Chapter 2. Binary search can still be implemented for action parts that do not subsume other action parts. Another possibility is to renumber the terminal symbols so that some subset of the very large action parts can use binary search and still subsume other action parts. Such a renumbering need not be visible to the software developer if provided with a well-designed lexical/syntactic interface. However, finding an appropriate renumbering is a complex problem and has not been pursued in this research.

3.3. Subsumption Graphs of LR Parsers

The subsumption relation is used to generate the information required to reduce the size of a hard-coded parser in a systematic manner. It is convenient to formulate the problem in terms of graph theory, as Dencker *et al.* do for their graph-colouring scheme [DDH 84] and Ichbiah and Morse do for compression of weak precedence matrices [IcM 70]. Consider the set of *LR* states of an *LR* parser as a set of vertices, as in a transition graph. The *subsumption graph* of the parser is constructed by taking subsumption relations between states as the edges of the graph. Thus, an edge (A, B) exists *iff* state A subsumes state B.

As defined, the subsumption graph has cycles that are due to equivalence classes of states. All states belonging to the same equivalence class can use the same "`ac..`" label in the hard-coded parser. It is convenient to coalesce each equivalence class into a "super-state," so that an *acyclic subsumption graph* is obtained. A set of equal states is represented by a single vertex in the acyclic subsumption graph.

Consider the set of eight states of a hypothetical *LR* automaton defined by Table 3.1, below. For clarity, distinct actions are defined by distinct lower case letters, and it is assumed that all states have the same default action (*e.g.*, "error").

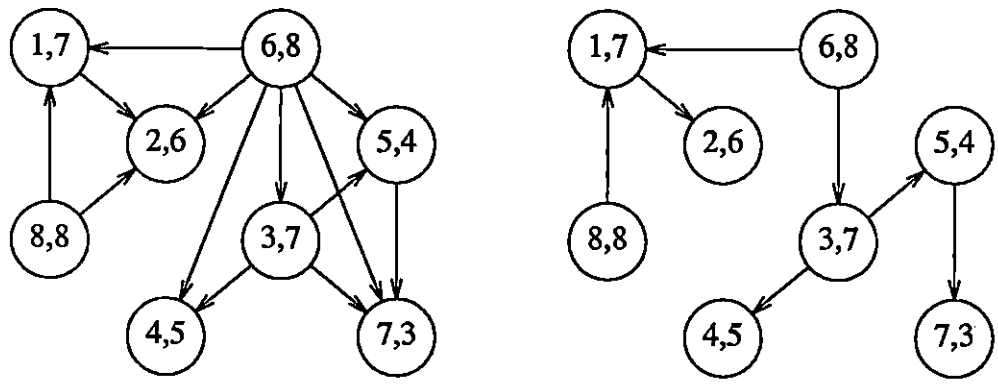
State	Weight	Actions	State	Weight	Actions
1	7	*bcdefgh*	5	4	*abcd*
2	6	*bcdfgh*	6	8	*abcdefgh*
3	7	*abcdefg*	7	3	*abd*
4	5	*acdfg*	8	8	*bcdefghi*

Table 3.1 : Hypothetical Set of States

The *weight* of a state indicates the number of non-default actions in that state. For example, the weight of state 14 in Figure 3.1 is five. Weights of zero are impossible in *LR* automata using shift-reduce actions. The sum of all weights of states that are not subsumed is directly related to the size of the actions part of a hard-coded *LR* parser.

The acyclic subsumption graph of the eight states in Table 3.1 is shown in Figure 3.2 (a). Vertices in the graph are labeled by state number, followed by state weight. Because of the transitivity of the subsume relation, subsumption graphs and acyclic subsumption graphs are *transitive.closures.* A convenient simplification is to consider the *transitive reduction* of the acyclic subsumption graph, which is shown in Figure 3.2 (b). There is no loss of information since there is a one-to-one correspondence between transitive closure graphs and transitive reduction graphs (these types of graphs and their properties are discussed in detail in [Meh 84]).

The acyclic subsumption graph of an *LR* automaton is found using the algorithm in Figure 3.3. From a theoretical viewpoint this algorithm is not optimal, but is sufficient for this research. The main idea is to avoid computing the subsumption relation when transitivity permits, and to avoid re-computing the subsumption relation between any two states. Ordering the states (vertices) by decreasing weight ensures that when comparing two states A and B such that A precedes B in the ordering, A equals B *iff* A subsumes B and the weights of A and B are equal, and B subsumes A *iff* A equals B. A related point is that any ordering by decreasing vertex weights of an acyclic subsumption graph, whether reduced or not, is a topological sort (the converse is not true). The subroutine *subsumes(i, j)* calculates the value of the relation between two states. It is trivial to implement and works in time linearly bounded by the weight



(a) Transitive closure (b) Transitive reduction

Figure 3.2 : Subsumption Graphs

of state i, assuming that the actions of states are ordered by the *LR* program used to generate the states.

The algorithm finds all the equivalence classes of states simultaneously with the acyclic subsumption graph. At completion, the value in *equalstates*[i] contains 0 if state i is in the acyclic subsumption graph, or n where n is the first vertex in the ordering which is equal to state i. The array *edges* is a bit map that ends up being the adjacency matrix representation of the transitive closure. Edges of the reduced closure graph are added at the line *edges*[i,j] := 1 , and transitive closure edges by the line *edges*[i,k] := 1 . Thus, if the reduced graph is desired, an adjacency list representation can be constructed from the first assignment instruction, and the *edges* array discarded upon completion of the algorithm.

In practice, it is desirable to consider for subsumption only those states with a certain minimum weight. The extra direct branch overhead is too large to be worthwhile for very small states. Also, the construction and subsequent analysis of the subsumption graph is greatly simplified by considering only the larger states. A minimum

```
procedure subsume_graph;
begin
        Initialize arrays equalstates and edges to 0;
        Sort states in order of decreasing weight;
        for i:=1 to num_states do for j:=i-1 downto 1 do
                if edges[i,j]=0 and equalstates[j]=0 then
                        if subsume(i,j)=TRUE
                                if weight(i)=weight(j)
                                        if equalstates[j]=0 then
                                                equalstates[i] := j
                                        else
                                                equalstates[i] := equalstates[j]
                                else begin
                                        edges[i,j] := 1;
                                        for k:=1 to j-1 do
                                                if edges[j,k]=1 then
                                                        edges[i,k] := 1
                                end {else}
end; {subsume_graph}
```

Figure 3.3 : Algorithm for Constructing Acyclic Subsumption Graph

weight of seven non-default actions per state is assumed in the analysis and results presented here. Naturally, the terminal and nonterminal actions of states are considered separately. Subsumption relations do not hold well for nonterminal transitions. Thus, nothing is gained by applying subsumption algorithms to the "goto" part of a parser. Also, most nonterminal transitions are eliminated using the methods of Chapter 5.

3.4. Inverse Arborescence

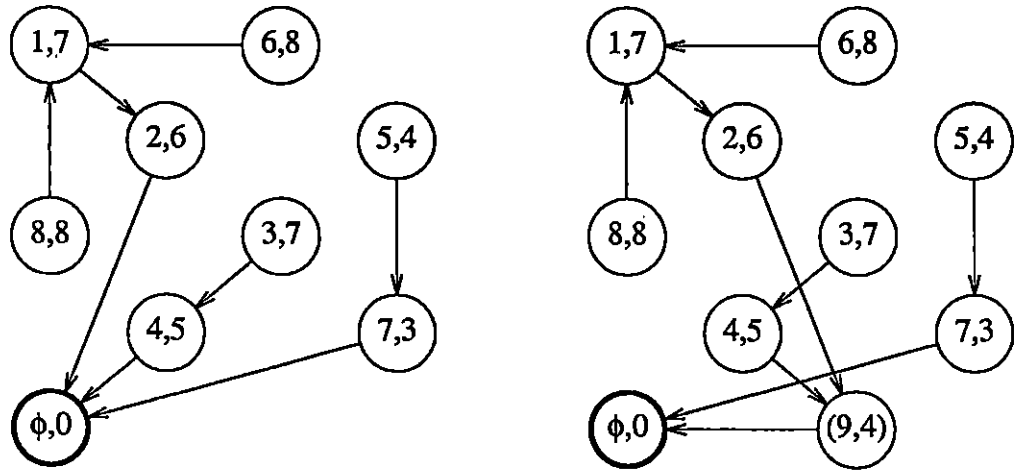
In 1970, when *LR* theory was still relatively undeveloped, the acyclic subsumption graph was defined and used by Ichbiah and Morse in order to compress weak precedence matrices to make small weak precedence parsers [IcM 70]. Their method involves assigning weights to graph edges, rather than vertices, and the addition of another vertex, ϕ, that is subsumed by all others. Since they worked with transitive

closure graphs, all vertices other than ϕ have an edge directed to ϕ. The weight of an edge (A,B) is defined as the difference between the weights of vertex A and vertex B. The meaning of an edge weight on (A,B) is the number of "read productions" that are in A but not in B, and thus gives a measure of size if A is chosen to subsume B without any intervening states.

Once the acyclic transitive closure subsumption graph with ϕ is constructed, an almost-optimal sharing of "actions" (*productions*, in [IcM 70]) is obtained by finding the minimum-cost *inverse arborescence* of the graph. An inverse arborescence of a graph is simply a *reverse spanning tree* of that graph; all vertices other than ϕ are constrained to have an outdegree of one. Such a subgraph must always exist for the graphs defined by Ichbiah and Morse, with ϕ as the "root." A minimum-cost inverse arborescence is easily obtained by choosing from each vertex A other than ϕ the arc out of A having the least weight. A discussion of correctness is included in [IcM 70]. Figure 3.4 (a) shows the minimum-cost inverse arborescence of the graph shown in Figure 3.2 (a). In this case, the cost is 20.

In their definitive paper on *LR* parsing, Aho and Johnson suggest that the methods of Ichbiah and Morse for compressing weak precedence parsers be applied to directly-executable *LR* parsers [AhJ 74]. However, an inverse arborescence is unsuitable for *fast* hard-coded parsers because some parse states might require many unconditional "goto" instructions. These instructions both occupy more space, and take time to execute. For example, the actions of a state could occur in several different parts of the parser, as in the following example:

```
      ...
st23: push(23);
ac23: if (tok==35) goto st45;
      if (tok==10) goto sr4;
      goto ac6;
```



(a) Minimum-cost inverse arborescence

(b) Minimum-cost after addition of fictitious state

Figure 3.4 : Inverse Arborescence

```
        ...
ac6:    if (tok==21) goto sr5;
        if (tok== 2) goto st34;
        goto ac78;

        ...
ac78:   if (tok== 14) goto st2;
        goto r6;
```

While parsing, if the lookahead symbol is 14 when in state 23, two unconditional branches are executed before the correct transition (to state 2) is found. In general, if a state is represented by vertex A, then the maximum number of goto instructions required to search through that state's actions is related to the number of vertices along the path $<A, \cdots, \phi>$ that have an indegree greater than one in the inverse arborescence, not including vertices A and ϕ. The example in Figure 3.4 (a) has only one such vertex (vertex 1), but the inverse arborescence formulation allows for many. The

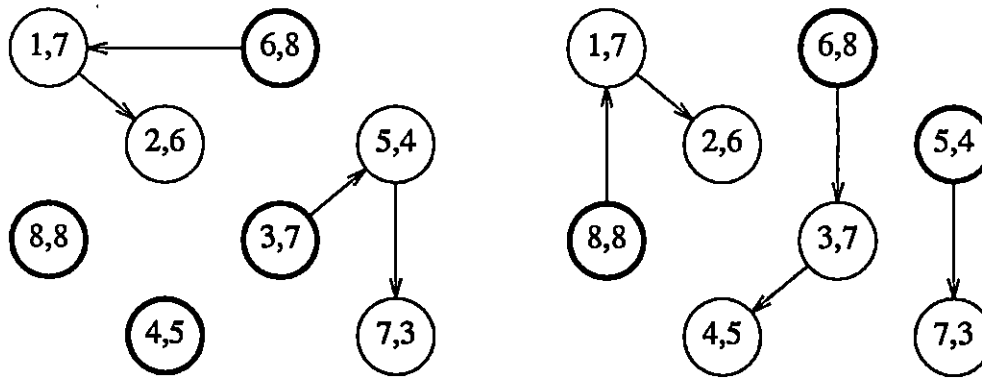
problem is that the formulation is too general.

Another problem with the inverse arborescence approach is that the inclusion of fictitious states in the acyclic transitive closure graph can lead to inverse arborescences of even smaller cost [IcM 70]. For instance, if a fictitious state 9 with actions *cdfg* is added to the graph of Figure 3.2 (a), an inverse arborescence of cost 16 is obtained, since states 2 and 4 can share the actions in "state" 9, as shown in Figure 3.4 (b). Deciding what fictitious states should be added is in general a complex problem.

3.5. Chain Decomposition

The problem of "spaghetti" branching to search through a state's actions is avoided by restricting the format of the hard-coded parser to the construction shown in Figure 3.1. That is, only one unconditional branch to access the actions is permitted, the branch must occur before any of the actions, and all actions must appear consecutively, immediately followed by the default action. This is accomplished by finding a *chain decomposition* of the acyclic subsumption graph (not the reduced graph). The *chain weight* is determined by the weight of the vertex at the head of the chain, because all the actions of the states corresponding to the vertices in the rest of the chain may be subsumed by the actions of the state corresponding to the first vertex. The total weight is the sum of all the chain weights involved in the decomposition, and represents the number of conditional branch instructions that are needed in the action parts of the compressed hard-coded parser. Note that a chain decomposition is a special case of an inverse arborescence, the constraint being that all vertices other than ϕ must have an indegree of either one or zero.

Figure 3.5 shows two different chain decompositions of the graph shown in Figure 3.2 (a). The chain heads are highlighted by heavy circles. For every edge (A, B) in the chain decomposition, vertex A is said to *directly subsume* vertex B. If there is a path $<A, \cdots, B>$ in the chain decomposition but no edge (A, B), vertex A *indirectly subsumes* vertex B. The decomposition in Figure 3.5 (b), with a total weight of 20 in three chains, is the only optimal one. It is identical to one of two optimal minimum-cost inverse arborescences, one of which is shown in Figure 3.4. The decomposition shown in Figure 3.5 (a), with a total weight of 28 in four chains, is sub-optimal for two



(a) Sub-optimal decomposition (b) Optimal decomposition

Figure 3.5 : Chain Decompositions

reasons: (1) vertex 6 is chosen to subsume vertex 1, leaving vertex 8 with nothing to subsume that couldn't already be subsumed by vertex 1, and (2) vertex 5 is chosen to be subsumed by vertex 3, but vertex 4 would have been a better choice since its weight is greater than that of vertex 3.

The algorithm shown in Figure 3.6 is used for finding chain decompositions of acyclic reduced subsumption graphs. It incorporates several heuristics that do not guarantee an optimal decomposition, but still provide one for each test grammar. First of all, the vertices of the graph are ordered by decreasing weight. The algorithm uses the intuitively sensible principle that for any vertex A, the heaviest vertex possible B should be chosen for direct subsumption. This heuristic solves the second problem mentioned in the previous paragraph, but there is still no guarantee that the vertices will be ordered such that vertex 8 instead of vertex 6 is chosen to directly subsume vertex 1 in Figure 3.5. To eliminate this problem, the vertices are also sorted by order of increasing outdegree within decreasing weight. However, it is still possible to make an incorrect choice. A third heuristic used is to choose for direct subsumption the vertex

```
procedure chain_decomp;
begin
        Order the graph's vertices by increasing outdegree within decreasing weight;
        for each vertex A in ordering do begin
                Choose the heaviest vertex B subsumed by A;
                if there is more than one vertex B then
                        choose B with lowest indegree;
                Retain the edge (A,B) for chain decomposition;
                Remove all other edges incident to B
        end {for}
end; {chain_decomp}
```

Figure 3.6 : Heuristic Chain Decomposition Algorithm

with the smallest indegree among those of equal weights. In fact, the idea of subsuming the heaviest possible vertex does not always lead to an optimal chain decomposition. Whether or not the problem is solvable in polynomial time is unclear.

3.6. Subsumption Variations

A possibility that has heretofore been ignored is to allow an unconditional branch to occur part way through a state's actions. In other words, the condition stated in Section 3.2 that all actions must occur after the unconditional branch is relaxed. This seems to be a sensible idea; after all there is still only one unconditional branch allowed per parse state. This idea has not been pursued for two reasons. First of all, the problem of optimal subsumption is greatly complicated, because the original definition of the subsumes relation becomes invalid. In fact, the new problem shows some similarity to the fictitious state problem discussed in Section 3.3.

The second reason why the original subsumption relation is best adhered to is that it is possible to eliminate some or all of the overhead associated with the unconditional branch leading to a state's actions. For example, in Figure 3.1 (b) state 23 is implemented via a branch to label `st23` or `sc23`, followed by the required "push"

instruction, and finally another branch to label `ac23`. State 23 could also be implemented by a combined "push-and-branch" or "push-scan-and-branch" instruction located in the action blocks of each of state 23's predecessor states. That is, instead of code of the form

```
ac9:  ...
      if (tok== 8) goto st23;
      ...
ac13: ...
      if (tok== 8) goto st23;
      ...
st23: PUSH(23);
      goto ac23;
```

the following construction is used:

```
ac9:  ...
      if (tok== 8) { PUSH(23); tok=scan(); goto ac23; }
      ...
ac13: ...
      if (tok== 8) { PUSH(23); tok=scan(); goto ac23; }
      ...
```

Thus, state 23 no longer exists *per se*, and the overhead for accessing label `ac23` appears to be only one conditional branch, just like all other states. However, there are several problems that must be considered. First of all, more space is required if state 23 has more than one predecessor, since every predecessor needs the push and scan instructions imbedded somewhere in its actions.

Another problem is that although the C code for the hard-coded parser might look like it will execute more quickly, there is no guarantee that the C compiler generates the expected object code. For instance, there might be subtle "pipeline" problems involved with compare-and-branch instruction lists, where some of these compare-and-branch instructions are actually compare-and-push-and-scan-and-branch. A final drawback to this approach is that the semantic code associated with any state eliminated in this manner must occur in the same location(s) as the push and scan instructions. If such instructions occur several times throughout the parser for a given eliminated state, the semantic code must also be repeated several times.

3.7. Compression Results

The algorithms shown in Figure 3.3 and Figure 3.6 were used to produce compressed fast hard-coded parsers for the five test grammars. The results are summarized in Table 3.2. The first section, "States," shows the total number of states, the number of equivalence classes of size greater than one, the number of states in the equivalence classes, and the number of states whose actions could be subsumed in those of another. The second section, "Transitions," gives the number of terminal transitions that would exist in the action blocks of a parser without using state subsumption, and the number of transitions that are eliminated through the use of state subsumption.

The overall improvement in the total sizes of the action parts range from approximately 60% for C to 32% for Oberon. The best improvements, both absolute and relative, are for those grammars with greater numbers of total terminal transitions, *i.e.* C, Pascal, and Algol. Note that few subsumption relations exist in the test grammars, because most of the eliminated transitions are due to equivalence classes.

Grammar	States				Transitions	
	total	classes	equal	subsumed	total	eliminated
C	181	6	46	5	1349	818
Pascal	203	4	41	2	707	366
Oberon	171	2	17	0	383	132
XPL	97	3	18	2	345	135
Algol	94	3	32	1	508	339

Table 3.2 : Subsumption Results

CHAPTER 4

MINPUSH PARSING

4.1. Introduction

One of the reasons that *LL* parsers typically run faster than their *LR* cousins is that the former use their stacks significantly less often. An *LL* parser pushes once for every new rule it begins to reduce, with a corresponding pop occurring after the rule has been completely recognized. (In the case of predictive recursive-descent parsers, these stack manipulations are performed implicitly via procedure call and return.) In contrast, an *LR* parser pushes once for every single shift transition that it performs, both terminal and nonterminal. Thus, there is more pushing (and popping) required in an *LR* parser than in an *LL* parser if any rule with more than one symbol on its right-hand side is involved in a derivation.

Of course, the benefit of greater recognition power justifies to some degree the extra stack use that an *LR* parser requires. However, a large fraction of this stack use is entirely redundant. Consider an *LL(1)* grammar that is used to construct one parser with an *LL(1)* parser generator, and another parser with an *LALR(1)* parser generator. Both of these parsers recognize the same language, yet the second almost always uses the stack more often than the first, and runs more slowly. Every time the slower parser is used, a price is paid for using too powerful a parsing method.

Theoretically, the stack used to help parse a context-free language should only be required when parsing *self-embedding* (or *nesting*) grammatical constructs. A self-embedding nonterminal symbol A has the property that $A \Rightarrow^{*} \alpha A \omega$, where α and ω are non-empty strings. If a derivation involves such a nesting, a stacking mechanism is required to balance occurrences of α with occurrences of ω. In fact, if a grammar does not generate any of these nesting constructs, it can be shown to be regular and

should therefore be able to be parsed without any use of a stack at all. The techniques of this chapter and the next show how this can be done within the *LR* framework.

There exist practical reasons for reducing use of the parse stack. Most importantly, parsers run faster, which is our primary goal. Not only are push instructions executed less often, but also the corresponding pop instructions. Secondly, hard-coded parsers that use the stack less often are smaller since the code required to push and pop occurs less frequently. This is especially significant for hard-coded *LR* parsers, where the instructions for pushing onto the parse stack, and perhaps checking for stack overflow, must normally occur once for each state. Finally, the amount of memory reserved for the stack need not be as large for reduced-push parsers.

4.2. Minpush-LR Parsing

The key to reduced pushing in *LR* parsers is contained in the following rather trivial observation: in the course of a parse only a fraction of all the *LR* parse states are ever uncovered at the top of the stack. These states are precisely those that are accessed by the parser in order to restart the parse after rules are reduced (*i.e.*, the *goto* action). If a parsing algorithm which stacks only these states can be devised, then quite a lot of redundant pushing can be eliminated. One such algorithm, called *minpush-LR parsing* (or *MLR* parsing), is developed in this chapter. The name does not mean to imply that pushing is kept to a minimum, but the algorithm is a natural first step in this direction.

The first problem in constructing a minpush parser is to determine the set of *push states* – the states that must be pushed onto the parse stack, because they might be subsequently accessed. State numbers in the parse stack are used as indices to rows of the "goto" table or, in the case of hard-coded *LR* parsers, as case labels that define goto blocks. Therefore, these states are those which have one or more nonterminal transitions out of them in the *LR* transition automaton. Now, if a state contains a transition on nonterminal N, its item set must contain at least one item of the form $[A \rightarrow \alpha \cdot N \beta]$. It follows that in the same state there must also exist one or more items of the form $[N \rightarrow \cdot \gamma]$, due to the closure operation performed on the previous item. The first item might be either a kernel item or a closure item (when $\alpha \equiv \varepsilon$), but the second must be a closure item. Since a closure item can only be created from an item of the form

$[A \rightarrow \alpha \bullet N \beta]$, where N is some nonterminal, the set of push states is simply the set of states that have one or more closure items.

At this point it is expedient to introduce some informal definitions. A *reduction path* is a path through an *LR* transition graph that "traces out" a rule reduction. The first state of a reduction path has a closure item of the form $[A \rightarrow \bullet X\ Y \cdots Z\ ]$ and a transition on symbol X to the second state of the reduction path, which contains the item $[A \rightarrow X \bullet Y \cdots Z\ ]$, and so on. The second item is called the *successor item* of the first, and the first item is a *predecessor item* of the second. The *LR* construction method guarantees that the successor of an item, if it exists, is unique. However, an item may have more than one predecessor item. An item has one or more predecessor items *iff* it is a kernel item, with the exception of the item $[S' \rightarrow \vdash \bullet S \dashv\ ]$ in the initial state, which has no predecessor. An *origin state* is a state that has one or more closure items. The name comes from the fact that reduction paths always originate in these states. Reduction paths terminate in *final* states, which have one or more *reduce items* of the form $[A \rightarrow \alpha \bullet\ ]$, or *shift-reduce items* of the form $[A \rightarrow \alpha \bullet X\ ]$ where the action on symbol X calls for a shift-reduction of rule $A \rightarrow \alpha X$. Leaves of an *LR* transition graph are always reduce or shift-reduce states, but the converse is not true, due to inadequate states with shift/reduce conflicts. An item has a successor item *iff* it is neither a reduce item nor a shift-reduce item.

Figure 4.1 shows the two reduction paths of the rule $E \rightarrow E+T$ within the *LR* transition graph of grammar G_1. Such reduction paths with common reduce states but differing origins are called *merged* reduction paths. The state where two reduction paths first merge is called a *merge state*. A state is a merge state *iff* it contains at least one item which has more than one predecessor. Therefore, a state is a merge state *iff* its indegree in the transition automaton is greater than one. In Figure 4.1, states 0 and 5 are the two origins for the rule $E \rightarrow E+T$, 3 is a reduce state, and 2 is a merge state. State 2 is also an origin for all rules with left-hand side T. Henceforth, origin states in automata diagrams are indicated by darkened circles, as shown by states 0, 2 and 5 in Figure 4.1.

If a rule p has n symbols on its right-hand side, the length of a reduction path of p is n if the path ends in an inadequate state, $n-1$ otherwise. (The second case occurs

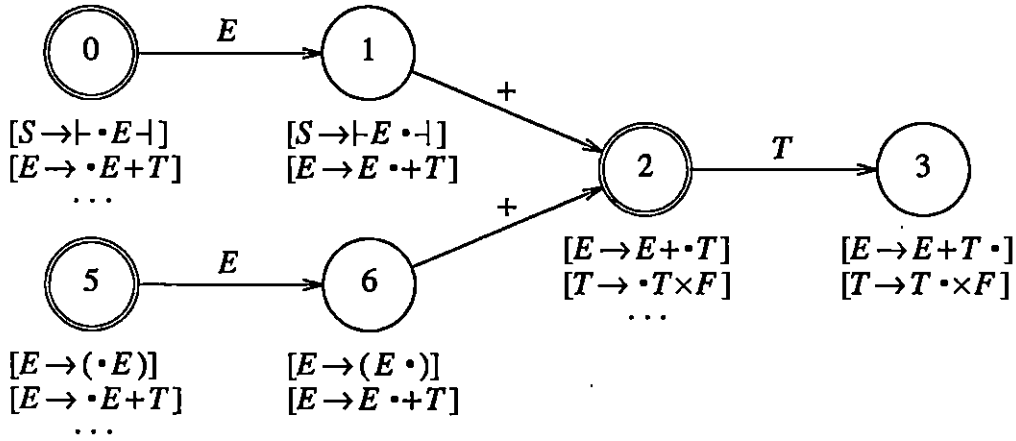


Figure 4.1 : Merged Reduction Paths

when the rule r is shift-reduced.) Reduction paths of zero length exist; they are created from *null* items and *unit shift-reduce* items. A null item has the form $[A \rightarrow \bullet \varepsilon]$. A unit shift-reduce item has the form $[A \rightarrow \bullet X]$, and the action on symbol X in the state containing the item is "shift-reduce by rule $A \rightarrow X$."

Now that the *LR* origin states have been identified, it is possible to devise a hypothetical parsing algorithm that pushes only these states. One would expect the operation of a minpush parser to be quite similar to that of an *LR* parser, and this is indeed the case. The only differences are:

(1) A minpush parser has both *push* and *shift* actions. A shift action does not push onto the stack, and a push action both shifts and pushes one state onto the stack.

(2) When reducing a rule, an *LR* parser pops the stack by the number of symbols on the right-hand side of the reduced rule. The corresponding *MLR* parser pops by an amount which can be no more, and is probably less, than the length of the right-hand side.

It remains to be seen how to compute the values of these *pop-counts*. Figure 4.2 shows the interpretive version of the minpush parsing algorithm, written in C.[1] The algorithm assumes that the transition tables are contained in two very sparse two-dimensional tables A and G (for *action* and *goto*, respectively), and that the left-hand side nonterminals of all rules are contained in the array LHS. Tokens are returned by calls to the *scan()* routine. Unlike the algorithm given in Figure 2.1, the "shift-reduce" action is used, indicated by the action code "SHR."

A minpush interpretive parser is slightly larger than the corresponding conventional *LR* parser, due to the new "PUSH" action and the few extra lines in the algorithm of Figure 4.2. The extra lines in the interpreter loop are insignificant since they occur only once in the entire parser. The effect of the additional "PUSH" action on parse table size depends on how the tables are factored (if at all), and how many bits are already required to encode the different actions. If the "ERROR" action is factored out into a separate table (as it often is in practice), the number of different actions to be encoded in the main table increases from four to five. In a straightforward implementation of the parse tables (allowing compaction of table rows and columns) this could cause an increase from two bits to three bits for the amount of memory required to encode the action.

There is another problem that could increase the size of an interpretive minpush parser. Recall that the conventional *LR* strategy requires an array containing the right-hand side lengths of all the production rules, so that the correct pop-count can be determined when a rule is reduced. Thus, the pop command of the minpush parsing algorithm would be replaced with

```
pop( LEN[action.rule] );
```

where LEN is the right-hand side length array. In the minpush parser, we do not yet know whether or not all reduction items of a particular rule require the same pop-count, since that rule may be reduced in several different states. A single LEN array indexed by rule numbers is insufficient if there is more than one pop-count for the same rule.

[1] The C language is used for this example because flow of control "falls through" switch case labels, which is convenient for PUSH and SHIFT, and SHR and REDUCE.

```
minparse()
{
        state = 0;          /* 0 is the start state  */
        push( 0 );          /* push the start state */
        token = scan();

        while (1) {

                action = A[ state, token ];
                switch ( action.code ) {

                 case PUSH:   push( action.state );
                 case SHIFT:  token = scan();
                              state = action.state;
                              break;

                 case SHR:    token = scan();
                 case REDUCE: do {
                                      reduce( action.rule );
                                      pop( action.popcount );
                                      lhs = LHS[rule];
                                      state = topstack();
                                      action = G[ state, lhs ];
                              } while (action.code == SHR);
                              state = action.state;
                              if (action.code == PUSH)
                                      push( state );
                              break;

                 case ERROR: error();

                 case HALT:  return;

                }
        }
} /* end minparse */
```

Figure 4.2 : Minpush Parsing Algorithm

Instead, the pop-count could be an additional attribute of the reduce or shift-reduce action, just as the rule number is an attribute. The pop-counts would then have to be encoded as part of the action table's reduce actions, or factored out into a table of their own. In the algorithm of Figure 4.2, the pop-count is considered an attribute of the reduce action (*i.e.*, *action.popcount*).

4.3. Direct MLR Construction

The minpush parsing algorithm is useless without minpush parsing tables. As already shown, it is easy to determine the *LR* states that should be pushed when entered – they are simply the origin states. The only remaining problem is to find the proper pop-count for every reduce and shift-reduce action. It turns out that these pop-counts can always be unambiguously determined, although the required minpush automaton is occasionally slightly larger than the corresponding *LR* automaton.

Consider the operation of the minpush parser given in Figure 4.2. A push onto the parse stack occurs with every transition into an origin state. Popping the stack occurs when a rule is reduced, and the pop is used to uncover some nonterminal's origin state at the top of the stack. However, an origin state might be the origin of reduction paths for more than one nonterminal. For this reason the origin state is not popped until it is *impossible* for it to be an origin of any of the rules currently being reduced. Thus, the pop-count for a reduce item is equal to the number of origin states that its reduction path passes through, subtracting one for the first origin state (the head of the path), which must remain on the stack until popped by another rule reduction higher up in the parse tree.

There are at least two different ways to find the pop-counts: a set of *LR(k)*-like states with items augmented with push information can be generated directly from the source grammar, or an existing set of *LR* tables (or states) can be adapted to be minpush parsable. These two methods are called *direct MLR construction* and *MLR construction by adaptation*, respectively. This section presents the direct construction method, which closely follows the *LR* method. The emphasis is on minpush automata built from *LALR(1)* grammars (*i.e.* those grammars parsable with an *LR(0)* set of states). Extension to *LR(k)* grammars is straightforward. The problem is to create a

deterministic minpush parsing automaton for an *LALR(1)* grammar $G = <V_t, V_n, S, P>$.

Definition: A *minpush-LR(0) item* (or *MLR(0) item*) is a quadruple $<A, \alpha, \beta, p>$, where the rule $A \rightarrow \alpha\beta$ is a member of the production set P, and p is an integer such that $0 \leq p \leq |\alpha|$, called the *push-count* of the item. We use the familiar "dot" notation with *MLR(0)* items, *e.g.*

$$[A \rightarrow \alpha \cdot \beta, p]$$

The *core* of the above item is simply the *LR(0)* part, $[A \rightarrow \alpha \cdot \beta]$. Naturally, an *MLR(k)* item is an *MLR(0)* item augmented with a lookahead string of length k, *e.g.*

$$[A \rightarrow \alpha \cdot \beta, p, \gamma]$$

where $|\gamma| = k$. Kernel items are those with nonempty α (*i.e.*, the dot is not at the far left) and nonkernel (closure) items are those with $\alpha \equiv \varepsilon$.

A complete set of *MLR(0)* states is generated much in the same way as they are in the *LR(0)* construction algorithm. Slightly modified *closure* and *goto* functions are required. The closure function is almost exactly the same as the closure function used in *LR(0)* construction, the only modification being that closure items are initialized with a zero push-count. In the following, KS is a kernel set of items, and IS is a complete (kernel plus closure) set of items.

closure $(KS) \equiv$ smallest set IS such that
1. $KS \subset IS$
2. $[A \rightarrow \alpha \cdot B\beta, p] \in IS$ and $B \rightarrow \gamma \in P$
   implies $[B \rightarrow \cdot\gamma, 0] \in IS$

The *goto* function is used to find all successor states of a set of items IS. As in *LR(k)* automata, states (item sets) are distinguished by distinct kernel sets. The *goto* function obtains the kernels of other states, where these kernels are distinguished by item cores, *and* their push-counts.

$$goto(IS,X) \equiv \begin{cases} \{[A \rightarrow \alpha X \cdot \beta, p+1] \mid [A \rightarrow \alpha \cdot X \beta, p] \in IS\} & \exists \text{ an item of the form } [B \rightarrow \gamma \cdot X\, Z\, \delta, q] \in IS, Z \in V_n \\ \{[A \rightarrow \alpha X \cdot \beta, p] \mid [A \rightarrow \alpha \cdot X \beta, p] \in IS\} & \text{otherwise} \end{cases}$$

The above definition may look somewhat cryptic, but is easily explained in English: items in a kernel set generated by the *goto* function have their push-counts incremented *iff* that kernel set generates at least one closure item.

From the definitions of the *MLR(0)* closure and goto functions, it is possible to establish upper and lower bounds for the push-counts of *MLR* items. Push-counts of closure items are initially zero, and are incremented by at most one for each shift of the marker through the item's right-hand side. Therefore, given an *MLR(0)* item $[A \rightarrow \alpha \cdot \beta, p]$, the upper bound for the value of p is simply the number of symbols in α. In kernel items, every time the marker dot is to the immediate left of a nonterminal symbol, the push-count is incremented. Therefore, the lower bound is given by the number of nonterminal symbols in α, minus one if $\alpha \Rightarrow^* B\gamma$, where B is a nonterminal symbol. In other words, the lower bound does not include the first symbol in α, if that symbol is a nonterminal. If $\beta \Rightarrow^* C\delta$ for some nonterminal C, the lower bound is one higher.

A complete set of *MLR(0)* states is obtained in the usual *LR* manner: by repeatedly applying the *closure* and *goto* functions to "incomplete" states (those states consisting of only a kernel set, *KS*) until no incomplete states remain [AhJ 74]. Since push-counts are incremented as a side effect of the *goto* function, but do not affect item cores, it is clear that the number of *MLR(0)* states is either equal to or larger than the number of *LR(0)* states for a given grammar. In fact, an *LR(0)* set of states can be obtained from an *MLR(0)* set by simply eliminating the push-counts and combining states with identical item cores. Two or more *MLR(0)* item sets with identical kernel item cores, but differing push-counts associated with those cores, are called minpush *split* states. An *MLR(0)* set of states that is larger than the corresponding *LR(0)* set of states contains at least two split states.

Claim: *The MLR(0) state generating algorithm always terminates.* Unique states are determined by unique *MLR(0)* kernel sets. Corresponding to each unique kernel core set, there is either one non-split state, or two or more split states. But the number of split states must be finite, since there is a finite number of push-counts that can be associated with an item core. Therefore, the number of *MLR(0)* states is finite, and eventually they are all generated.

Theorem 4.1: *The set of LR(0) grammars and the set of MLR(0) grammars are equal.* Both an *LR(0)* set of states and an *MLR(0)* set of states can be generated from a grammar. The only differences between the *LR(0)* item sets and *MLR(0)* item sets is that the latter include push-counts, and there could be more than one *MLR(0)* state with the same item cores as a given *LR(0)* state. It follows that if no inadequate states exist in an *LR(0)* set of states, they do not exist in the *MLR(0)* set of states generated from the same grammar. Similarly, if no inadequate states exist in an *MLR(0)* set, they do not exist in the *LR(0)* set, since the *LR(0)* set can be obtained from the *MLR(0)* set by coalescing states with the same item cores. □

If there are one or more inadequate states in an *LR(0)* (or *MLR(0)*) set of states, lookahead sets are required to resolve potential shift/reduce and reduce/reduce conflicts. The most common lookahead sets used for this purpose are those generated by the *SLR(1)* and *LALR(1)* algorithms, as discussed in Chapter 2. If potential conflicts in an *MLR(0)* set of states are resolvable by using *SLR(1)* or *LALR(1)* lookahead sets, the grammar used to generate the set of states belongs to the *minpush-SLR(1) (MSLR(1))* class or the *minpush-LALR(1) (MLALR(1))* class, respectively.

Theorem 4.2: *The set of SLR(1) grammars and the set of MSLR(1) grammars are equal.* The *SLR(1)* algorithm finds lookahead sets for rules from an analysis of the grammar, independently of how a set of *LR* states is generated from that grammar. Since *LR(0)* and *MLR(0)* state sets are so similar for a given grammar, potential shift/reduce and reduce/reduce conflicts are the same in both state sets. Thus, potential conflicts are resolved for both state sets, or for neither. □

Theorem 4.3: *There exists a deterministic MLR(k) parsing automaton for every LR(k) grammar, for all* $k \geq 1$. If an *LR(k)* automaton is deterministic, there do not exist any shift/reduce or reduce/reduce conflicts in the inadequate states, because lookaheads of potentially conflicting items are disjoint [ÀhJ 74]. For every *LR(k)* state there exist one or more *MLR(k)* states that have the same item cores, and either the same item lookaheads or proper subsets of the same lookaheads. Thus, if the *LR(k)* states do not contain conflicts, neither do the *MLR(k)* states. □

Figure 4.3 shows the *MLR(0)* sets of items and associated automaton created for

I_0: $[S \rightarrow \vdash \bullet E \dashv, 1]$
$[E \rightarrow \bullet E + T, 0]$
$[E \rightarrow \bullet T, 0]$
$[T \rightarrow \bullet T \times F, 0]$
$[T \rightarrow \bullet F, 0]$
$[F \rightarrow \bullet (E), 0]$
$[F \rightarrow \bullet num, 0]$

I_1: $[S \rightarrow \vdash E \bullet \dashv, 1]$
$[E \rightarrow E \bullet + T, 0]$

I_2: $[E \rightarrow E + \bullet T, 1]$
$[T \rightarrow \bullet T \times F, 0]$
$[T \rightarrow \bullet F, 0]$
$[F \rightarrow \bullet (E), 0]$

I_3: $[E \rightarrow E + T \bullet, 1]$
$[T \rightarrow T \bullet \times F, 0]$

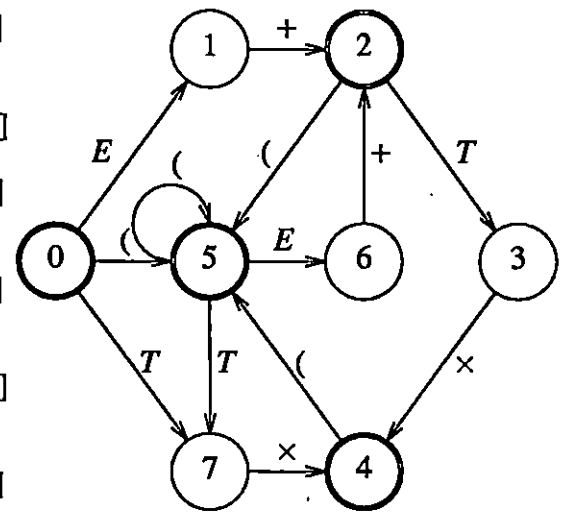
I_4: $[T \rightarrow T \times \bullet F, 1]$
$[F \rightarrow \bullet (E), 0]$
$[F \rightarrow \bullet num, 0]$

I_5: $[F \rightarrow (\bullet E), 1]$
$[E \rightarrow \bullet E + T, 0]$
$[E \rightarrow \bullet T, 0]$
$[T \rightarrow \bullet T \times F, 0]$
$[T \rightarrow \bullet F, 0]$
$[F \rightarrow \bullet (E), 0]$
$[F \rightarrow \bullet num, 0]$

I_6: $[F \rightarrow (E \bullet), 1]$
$[E \rightarrow E \bullet + T, 0]$

I_7: $[E \rightarrow T \bullet, 0]$
$[T \rightarrow T \bullet \times F, 0]$

(a) Sets of *MLR(0)* items



(b) Transition graph

Figure 4.3 : *MLR(0)* Sets of Items and Automaton

grammar G_1. Origin states in the *MLR(0)* automaton are indicated by bold circles. In *LR(0)* automata constructed from typical grammars, origin states account for about half of all *LR(0)* states, excluding *LR(0)* reduce-only states. Since *SLR(1)* lookahead sets are sufficient to resolve all *LR(0)* shift/reduce conflicts, the parser represented in Figure 4.3 is *MSLR(1)*. For grammar G_1, the *MLR(0)* states are the same as the *LR(0)* states. The next section shows that this is not always the case, due to the occurrence of split states. Unlike *LR(k)* and *SLR(1)* grammars, it turns out that the set of *LALR(1)* grammars and the set of *MLALR(1)* grammars are *not* equal, and this too is shown in the next section.

4.4. MLR Construction by Adaptation

The previous section showed that a minpush parser can be created for any *LR* grammar. The construction method is so similar to the *LR* method that it is also possible to create a minpush parser from an existing *LR* automaton. This method was used for most of the research described in this thesis, since an *LALR(1)* parse table constructor was readily available. To distinguish the new method from the constructor method of the previous section, the new method is called *minpush construction by adaptation.* It is shown that the two minpush constructor methods create isomorphic parse automata.

The starting point for constructing an *MLALR(1)* parser by adaptation is an *LR(0)* transition automaton, complete with item sets. The automaton can be extracted while performing conventional *LALR(1)* construction, or it can be reconstructed from the grammar and *LALR(1)* parse tables using the algorithm shown in Figure 4.4. Note that lookahead sets generated by the *LALR(1)* algorithm to resolve potential conflicts do not need to be regenerated. Such information is already within the transitions (actions) of the automaton. The item regeneration algorithm works for all *LR(k)* automata, as long as the original automaton is free of conflicts.

Once the *LR(0)* items have been obtained, they are augmented with push-counts, which are driven forward along reduction paths through the *LR(0)* automaton (in contrast to the reconstruction of item sets, which causes items to be propagated backward through the automaton). Push-count augmentation is performed using the algorithm of Figure 4.5. It occasionally happens that a phenomenon called *push-count conflict*

```
procedure additems;
begin
        for each state s do
                for each reduce action j in state s do begin
                        i := reduce item of j.rule;
                        additem( i, s );
                end
end; {additems}

procedure additem( i :ITEM; s :STATE )
begin
        if state s does not contain item i then begin
                add item i to state s;
                if item i is a kernel item then
                        for each predecessor state t do
                                additem( predecessor(i), t )
        end
end; {additem}
```

Figure 4.4 : Algorithm for Reconstruction of *LR* Automata

occurs, when the push-count of an item cannot be uniquely determined. This can happen only on merging reduction paths, and is first detected at merge states. This accounts for the condition test

if push-count of $i = p$ **then** $\cdots$

in procedure *augmentitem* of Figure 4.5. If an item has been augmented already, the item has been visited previously on some other reduction path. If this item's previously assigned push-count is the same as the current push-count, successor items in the reduction path have already been augmented (with the correct push-counts), and the augmentation for that particular path can terminate. If the previous push-count is *not* the same, there is a conflict for every remaining item in the reduction path. If the algorithm encounters no conflicts, the automaton can be used to minpush parse, with the push-counts of reduce items and shift-reduce items used for pop-counts.

```
procedure augment;
begin
        for each state s do
                for each closure item i in state s do
                        augmentitem( i ,0 );
end; {augment}

procedure augmentitem( i : ITEM; p : PUSHCOUNT );
begin
        if i has already been augmented then
                if push-count of i = p then
                        return
                else
                        augment item i with conflict;
        if item i has a successor j then begin
                if state of j is a push state then
                        p := p +1;
                augmentitem( j,p )
        end
end; {augmentitem}
```

Figure 4.5 : Push-Count Augmentation Algorithm

A state that contains one or more items with push-count conflicts is called a *push-count conflict state*. The easiest solution for resolving push-count conflicts is to simply *split* conflict states, much in the same way as some incremental *LR(1)* generating algorithms split *LR(0)* states until action conflicts are resolved [Pag 77]. Splitting states to resolve push-count conflict is the final stage in adaptive *MLR* construction, and is achieved using the algorithm of Figure 4.6. If a conflict state has a predecessor that is itself a conflict state, then that predecessor is split first, recursively. Thus, when a conflict state is split, it is possible to relink all shift transitions into the original unsplit state to the appropriate new split states. The correct destination for a given transition is found by considering the push-counts of kernel items in the predecessor states, and their

```
procedure resolvepushes();
begin
        while there is a state s with a conflict item i do
                statesplit( s, i );
end; {resolvepushes}

procedure statesplit( s :STATE; i :ITEM );
begin
        for each predecessor j of item i do
                if j is a conflict item then begin
                        t := state containing item j;
                        statesplit( t, j )
                end;
        Create a new state s';
        Link predecessors of s to s and s';
        Link successors of s to s'
end; {statesplit}
```

Figure 4.6 : Algorithm for Splitting Conflict States

successor items in the split states.

Claim: *The state-splitting algorithm always terminates.* This follows intuitively from the fact that there is a finite number of different push-counts that an item may have, and thus a finite number of split states that may be generated from an *LR* state.

Claim: *For a given grammar, the conflict states produced by adaptive construction have the same kernel cores as split states created by direct construction.* When direct construction generates a new *MLR(0)* kernel that has the same kernel core as a previously generated kernel, but different push-counts, this is the same situation as when adaptive construction locates a state with a push-count conflict. The direct construction method makes the new split state immediately, and the adaptive construction method makes the new state when performing the state-splitting algorithm.

Theorem 4.4: *Given an LR(k) grammar for any k, the direct and adaptive construction algorithms create isomorphic parsing automata.* Section 4.3 shows that *LR(k)* and directly constructed *MLR(k)* automata are identical, except that the latter sometimes have two or more split states that correspond to only one state in the former. The previous claim shows that the split states created by direct construction are identical to split states created by adaptive construction. Thus, the two construction methods generate the same set of states. □

Figure 4.7 shows a simple grammar (G_2) that generates a push-count conflict, its

I_0: $[S \rightarrow \vdash \bullet E \dashv, 1]$
$[E \rightarrow \bullet EA, 0]$
$[E \rightarrow \bullet A, 0]$
$[E \rightarrow \bullet B, 0]$
$[A \rightarrow \bullet abc, 0]$
$[B \rightarrow \bullet aC, 0]$

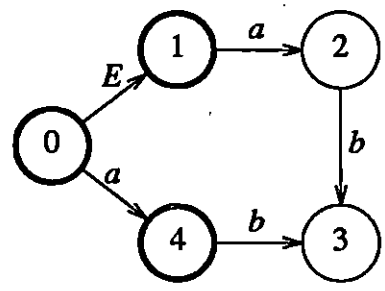
I_1: $[S \rightarrow \vdash E \bullet \dashv, 2]$
$[E \rightarrow E \bullet A, 1]$
$[A \rightarrow \bullet abc, 0]$

I_2: $[A \rightarrow a \bullet bc, 0]$

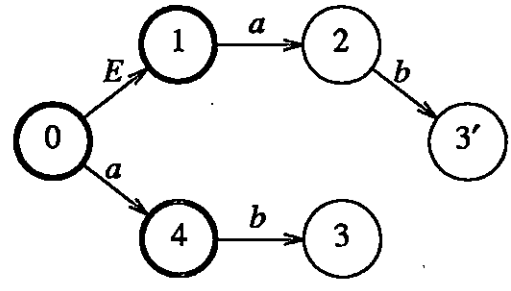
I_3: $[A \rightarrow ab \bullet c, 0/1]$

I_4: $[A \rightarrow a \bullet bc, 1]$
$[B \rightarrow a \bullet C, 1]$
$[C \rightarrow \bullet c, 0]$

(a) Sets of items for grammar G_2



(b) *LR*(0) automaton with conflict



(c) *MLR*(0) automaton

Figure 4.7 : Conflict Grammar G_2 and Automata

LR(0) automaton, and the *MLR(0)* automaton that results from applying the state splitting algorithm. State 3 is the only conflict state in the *LR(0)* automaton, and one split suffices to resolve the conflict, resulting in the two split states 3 and 3′. This is called a *split of degree two*. In all five test grammars of programming languages, split states occur rarely, and the degree of a split is never greater than two.

Claim: *Every LALR(1) grammar is also MLALR(1).* The construction by adaptation algorithm guarantees that the set of origin states for a reduce item in an *MLR(0)* set of states is a subset of the set of origin states for the reduce item in the corresponding *LR(0)* set of states. Lookahead sets generated by the *LALR(1)* algorithm are based on the origins of reduce items [DeP 79]. Thus, *LALR(1)* lookahead sets in *MLR(0)* automata are subsets of lookahead sets in corresponding *LR(0)* automata. If no action conflicts exist in an *LR(0)* automaton, they do not exist in the corresponding *MLR(0)* automaton.

Theorem 4.5: *The set of LALR(1) grammars is a proper subset of the MLALR(1) grammars.* If an *LR(1)* grammar is not *LALR(1)*, it is due to one or more reduce/reduce conflicts in the *LR(0)* set of states [FiL 88]. Such a grammar is *MLALR(1)* if *MLR(0)* construction splits states with reduce/reduce conflicts, due to *push-count* conflict, in such a way that the reduce/reduce conflicts disappear. This is illustrated by grammar G_3, below.

$S \rightarrow a\ A\ a \mid a\ B\ b \mid a\ x\ C \mid b\ A\ b \mid b\ B\ a$
$A \rightarrow x\ y$
$B \rightarrow x\ y$
$C \rightarrow x$

The reduced *MLR(1)* set of states for G_3 is shown in Figure 4.8. The items are *MLR(1)*, because they include both push-counts and lookaheads. The *MLR(0)* set of states is easily obtained by eliminating the lookaheads. It is not possible to coalesce states 10 and 11, because doing so would create a push-count conflict state. An *LALR(1)* parser generator would create only one state for states 10 and 11, because of their common core set, $\{\ [A \rightarrow x\ y\ \cdot\ ],\ [B \rightarrow x\ y\ \cdot\ ]\ \}$. The *LALR(1)* lookahead sets for both final items would be $\{\ a\ ,b\ \}$, resulting in a reduce/reduce conflict. But G_3 is *MLALR(1)*, because

I_0: $[S' \rightarrow \vdash \cdot S \dashv, 0, \varepsilon]$
$[S \rightarrow \cdot a\ A\ a, 0, \dashv]$
$[S \rightarrow \cdot a\ B\ b, 0, \dashv]$
$[S \rightarrow \cdot a\ x\ C, 0, \dashv]$
$[S \rightarrow \cdot b\ A\ b, 0, \dashv]$
$[S \rightarrow \cdot b\ B\ a, 0, \dashv]$

I_1: $[S \rightarrow \vdash S \cdot \dashv, 0, \varepsilon]$

I_2: $[S \rightarrow a \cdot A\ a, 1, \dashv]$
$[S \rightarrow a \cdot B\ b, 1, \dashv]$
$[S \rightarrow a \cdot x\ C, 1, \dashv]$
$[A \rightarrow \cdot x\ y, 0, a]$
$[B \rightarrow \cdot x\ y, 0, b]$

I_3: $[S \rightarrow b \cdot A\ b, 1, \dashv]$
$[S \rightarrow b \cdot B\ a, 1, \dashv]$
$[A \rightarrow \cdot x\ y, 0, b]$
$[B \rightarrow \cdot x\ y, 0, a]$

I_4: $[S \rightarrow a\ A \cdot a, 1, \dashv]$

I_5: $[S \rightarrow a\ B \cdot b, 1, \dashv]$

I_6: $[S \rightarrow a\ x \cdot C, 2, \dashv]$
$[A \rightarrow x \cdot y, 1, a]$
$[B \rightarrow x \cdot y, 1, b]$
$[C \rightarrow \cdot x, 0, \dashv]$

I_7: $[S \rightarrow b\ A \cdot b, 1, \dashv]$

I_8: $[S \rightarrow b\ B \cdot a, 1, \dashv]$

I_9: $[A \rightarrow x \cdot y, 0, b]$
$[B \rightarrow x \cdot y, 0, a]$

I_{10}: $[A \rightarrow x\ y \cdot, 1, a]$
$[B \rightarrow x\ y \cdot, 1, b]$

I_{11}: $[A \rightarrow x\ y \cdot, 0, b]$
$[B \rightarrow x\ y \cdot, 0, a]$

Figure 4.8 : *MLR(1)* Item Sets for a Non-*LALR(1)* Grammar

the *MLR(1)* machine is identical to the *MLR(0)* machine with items augmented with *LALR(1)* lookahead sets. Thus, G_3 is *MLALR(1)*, but not *LALR(1)*. Since every *LALR(1)* grammar is *MLALR(1)*, but not every *MLALR(1)* grammar is *LALR(1)*, the set of *MLALR(1)* grammars is larger than the set of *LALR(1)* grammars. □

There are alternatives to the state-splitting method of resolving push-count conflict. One idea is to declare some non-origin states as push states in such a way as to balance the push-counts along merging reduction paths. The main disadvantage of this idea is the complexity in determining these additional push states in such a way that other reduction paths passing through them do not themselves become conflict paths. Also, the speed of the resulting parser is slightly compromised. The state-splitting method appears to be the best because parsing speed is unaffected, the algorithm is fairly simple to implement, and the algorithm is invisible to the parser developer. The cost in space is marginal because split states are few in number, and, if they have large

action blocks, they are almost always subsumable.

The construction by adaptation algorithm is quite easy to implement given an existing *LR* parser generator, or a set of tables from such a generator. In the next sections we discuss how the *MLR* parsing method can be hard-coded, resulting in parsers that are always faster and usually smaller than the corresponding conventional *LR* hard-coded parsers.

4.5. Hard-Coding of Minpush Parsers

The minpush parsing algorithm is well-suited to hard-coding, because varying pop-counts are not so problematic, and especially because the structure of a hard-coded *LR* parser is easily adapted to accommodate the changed pushing. An important point about the minpush parsing algorithm is that pushing occurs when a push state is *entered*, rather than exited. Thus, a push instruction can be made the first part of the action block for each push state, rather than a side effect of every push transition. In a hard-coded minpush parser, the PUSH action is exactly the same as a shift action in an *LR* hard-coded parser: the current state is pushed onto the stack upon entry to that state. The minpush SHIFT action is manifested as simply the *absence* of the expected stack push. The result is that a hard-coded minpush parser requires a push instruction only once for every origin state, where the *LR* parser requires them for every state. Thus, the hard-coded minpush parser is smaller and runs faster.

The pop-counts for grammar rules cannot always be factored out in the same way as the push actions. In Chapter 2 we saw that the code for semantic actions associated with any rule reduction in a hard-coded *LR* parser need occur only once, and it is desirable to retain this property. If there are some rules that have more than one distinct pop-count, the pop cannot simply be put at the end of the semantic code for that rule. A simple alternative is to branch to a piece of code that performs the pop *before* going to the reduce block proper. For example, part of a hard-coded *MLR* parser might look like:

```
st15: if (tok==23) goto st9;
      if (tok==26) goto r45a;
      if (tok==78) goto st34;
      ...
st45: if (tok==26) goto r45;
      ...

r45a: sp -= 2;
      goto R45;
r45:  sp -= 1;
R45:  /* code for reducing rule 45 */
...
```

With this construction, the most common push-count for rule 45 is factored out and placed at the head of the reduction code (*i.e.*, at label `r45`), thus keeping the amount of extra branching to a minimum. If there are only two different pop-counts, the following construction is used:

```
r45a: sp -= 1;
r45:  sp -= 1;
R45:  /* code for reducing rule 45 */
```

This time, less code is generated. Execution time for branching to label `r45a` is only incidentally more than branching to label `r45`. The stack pointer is merely decremented twice instead of performing an extra branch.

It turns out that rules with non-unique pop-counts occur rarely in *MLALR(1)* grammars. Of course, they always exist for rule reductions associated with states that must be split due to push-count conflict. They also exist for other rules whose reduction paths are already split by *LR(0)* construction, but this is still more rare. Interestingly enough, rules that can be both reduced and shift-reduced, several of which occur in two of the five test grammars, do not ever have the non-unique pop-count problem. None of the test grammars have non-unique pop-counts that differ by more than one for a given rule.

A minor space optimization becomes possible with the smaller pop-counts. Usually, popping the stack occurs within the reduce block, then the parse is restarted by branching to the switch statement. Popping by zero is required for many rules, and is of

course done by having no pop instruction. Popping by one is the next most common, and the instruction to decrement the stack pointer is factored out and placed just prior to the switch statement, as in

```
swpop:      sp -= 1;
sw:   switch ( *sp ) {
      ...
```

Then, those reduce blocks that require a pop of one simply branch to the label `swpop`, instead of `sw`.

The use of state subsumption to compact terminal transitions, as presented in Chapter 3, is sometimes affected in hard-coded *MLR* parsers, due to non-unique pop-counts. Reduce and shift-reduce actions for these rules must be considered distinct actions for each pop-count. At first glance it might appear that subsumption would be more powerful with hard-coded *MLR* parsers, since no extra branch would be required to access subsumed states that do not require a push instruction. Unfortunately, this is not the case, because subsumable and equal states are always push states. States that are subsumable have many terminal transitions, and many terminal transitions are due to the presence of closure items.

Semantic actions can be inserted anywhere in an *MLR* parser where the corresponding *LR* parser would permit the same actions, because of the similarity of their underlying automata. Some existing parser generating systems permit semantic information to be stacked in parallel with the parse stack. YACC [Joh 75] uses this method, and Purdom and Brown suggest a similar mechanism for their *ELR(k)* parsers [PuB 81]. Unfortunately, in general it is not possible to guarantee that the minpush parse stack is used precisely when some semantic information should be stored. The fact that some rules might have more than one pop-count associated with their reduction introduces further complication. In short, using the parse stack for semantic analysis is unsuitable for minpush parsers.

Semantic analysis need not be performed using the YACC method. Many existing parser generator systems associate semantics with the reduction of *null nonterminals*, also called *semantic hooks* [PuB 80, Tho 77]. A null nonterminal A appears as the

left-hand side of only one rule, of the form $A \rightarrow \varepsilon$. They are used to associate semantic actions with points "inside" larger rules. Purdom and Brown discuss where and under what conditions semantic hooks can be used with *LR* grammars [PuB 80]. But use of null productions in this way increases the number of origin states, since they become closure items in the states where they might be reduced. Thus, some of the efficiency gained by using an *MLR* parser is lost. Of course, it is desirable to bypass the actions involved in reducing the null nonterminal, and go straight to the semantic code. Luckily this is always possible for origin states whose closure items are null items. Such issues are discussed in Chapter 5.

If a parser is used in a batch processing environment, some kind of syntax error recovery mechanism is desirable. Error recovery for minpush parsers is not rigorously pursued here, but some useful observations can serve as a first step in that direction. The minpush parse stack and the current state of a minpush parser contain the entire history of a parse, but in a slightly different format than in *LR* parsers. However, *LR* recovery algorithms can be applied to minpush parsers because the minpush stack can be translated to an equivalent *LR* stack. Corresponding to each origin state in an *MLR* parser there is exactly one state in the related *LR* parser. Adjacent entries in the *MLR* stack always define a unique path through the *MLR* automaton. The same claim can be made for the current state and the entry at the top of the *MLR* stack. Thus, in order to directly use *LR* recovery methods in *MLR* parsers, translation data structures can be used to convert from one stack form to the other.

A minpush stack bears some resemblance to an *LL* parse stack as well. Entries in the *MLR* stack are origin states. For every entry in the minpush stack there are one or more entries in the *LL* stack. Thus, it seems possible that an *LL*-based error recovery method is suitable for minpush parsers. Some recovery algorithms are applicable to both *LL* and *LR* parsers. For example, the method used by Gray in his DEER *LL* parser generator [Gra 87], and the method used by Dencker *et al.* in their PGS *LR* parser generator [DDH 84], is based on a method by Röhrich [Röh 80]. It is probable that such a method can be modified to work with minpush parsers, which lie somewhere between *LL* and *LR* parsers.

4.6. Checking for Stack Overflow

Reduced use of the parse stack does not eliminate the need to ensure that the parse stack does not overflow. The easiest way to do this is to include a stack height check on some subset of the pushing states. This is Pennello's approach for assembly-coded *LR* parsers [Pen 86]. However, the problem of finding a minimal or near-minimal set of checking states in an *MLR* parser is slightly changed due to the reduced number of push states. The proposed method for *MLR* parsers is similar to Pennello's method, but has a few important refinements.

First of all, we define the *push graph* of an *MLR* automaton. The vertices of the push graph are simply the push states of the automaton. For any two vertices A and B, a directed edge (A,B) is constructed *iff* there is a path from state A to state B in the *MLR* transition graph that does not pass through any other push states. The push graph shows the order in which stack pushes might be made while parsing. Cycles in the push graph represent recursive reduction paths, due to direct or indirect right recursion or self-embedding, that can cause the stack to grow without bound. For an *LR* parser, the push graph is simply the *LR* automaton, since every state is pushed. An *MLR* push graph typically has only half as many vertices.

The problem is to find a subset of the push states (*i.e.*, a subset of the push graph's vertices) in which stack overflow checks should be inserted. This subset is called the *checking set.* The overflow checks must guarantee that the stack cannot grow by more than some constant before another check is made. Each push state in the checking set is called a *checking state.* Since cycles can cause the stack to overflow, the problem becomes one of finding a subset of vertices which, if removed from the push graph, makes it acyclic. Obviously, we would like the checking set to be as small as possible.

The code for the beginning of a push state accessed by a nonterminal and requiring a stack check looks as follows:

```
st13: /* no scan call required */
      if (sp >= (maxstack-4)) goto stackerr;
      *sp++ = 13;
ac13: if (tok==12) goto st45;
      ...
```

Here, `maxstack` is the address of the maximum height of the stack, and `plen` is the maximum path length between a checking state and the next checking state. The code at label `stackerr` either halts the parser (in which case a simple branch instead of a procedure call is sufficient), or takes care of allocating more memory, copying the stack contents, redefining `sp` and `maxstack`, and so on. If the parser simply halts, then `maxstack` is a constant. Since `plen` is also a constant, though different for each stack check, a good C compiler generates a simple register-constant compare instruction to implement the check.

Pennello suggests that a polynomial time algorithm exists that finds a minimum checking set, but the problem is in fact intractable. In graph theory, the problem is called FEEDBACK VERTEX SET, and is proven NP-complete by reduction from a known NP-complete problem [GaJ 79].

> FEEDBACK VERTEX SET problem: Given a directed graph $G=(V,E)$ and a positive integer $k \leq |V|$, is there a subset $V' \subset V$ such that V' contains at least one vertex from every directed cycle in G?

Theorem 4.6: *FEEDBACK VERTEX SET is NP-complete.* The proof is by showing that the well-known VERTEX COVER problem reduces to a special-case FEEDBACK VERTEX SET problem via a reversible transformation, computable in polynomial time. The vertex cover problem is to find a subset of vertices $V' \subset V$ of size $k \leq |V|$ in an undirected graph $G=(V,E)$ such that for every edge $(a,b) \in E$, either $a \in V'$ or $b \in V'$. Now consider a transformed digraph $G'=(V,E')$. For every undirected edge $(a,b) \in E$, there exist two directed edges (a,b) and (b,a) in E'. Thus, $|E'| = 2 \times |E|$. It is clear that such a transformation is reversible, and can be completed in linear time. It should also be clear that the vertex cover problem in G is equivalent to the feedback vertex set problem in G'. Since VERTEX COVER is NP-complete, FEEDBACK VERTEX SET is also NP-complete. □

Pennello notes that each *strongly-connected component* (SCC)[2] of an *LR* transition graph can be considered separately. Since Pennello doesn't illustrate his "polynomial time" algorithm, he uses a heuristic to find the checking states. In short, he divides the *LR* automaton into SCC's, then for each SCC randomly removes a vertex until the SCC becomes acyclic. The algorithm used to find the checking set for an *MLR* parser, shown in Figure 4.9, avoids finding SCC's and uses a heuristic that works much better in practice. The algorithm is based on depth first search (DFS) of the push graph. Edges encountered in a DFS can be partitioned into four classes: tree, forward, cross, and back edges. It is well-known that every cycle in a digraph includes at least one back edge [Meh 84]. Thus, removal of one vertex from each back edge guarantees that all cycles are broken. It is a subset of these vertices that is chosen for the checking set (given by the bit vector *checkset* in Figure 4.9).

The algorithm works by "pruning" the DFS search tree at vertices that are found to be the heads of back edges, so that cycles already broken by one checking state are not subsequently re-broken by another. This is why a search through all of the edges leaving a vertex is done before DFS proceeds from that vertex – if any back edge is found, the current DFS is halted. Thus, for each checking state found by the algorithm there is at least one cycle in the push graph that includes this checking state and no others. In this sense, the checking set found is *minimal*, but not necessarily *minimum*. Obviously, the number of states in the checking set found by the algorithm of Figure 4.9 is less than or equal to the number of back edges found in a conventional DFS.

In a hard-coded *MLR* parser, stack checks are implemented as comparisons between the stack pointer and a value determined from the local constant `plen` and the pointer to the top of the stack, `maxstack`. The value `plen` need not be the same for each checking state, which is another point Pennello seems to have missed. He uses a common checking routine, called from each checking state before a push onto the stack is done. This routine always makes sure that there is enough room on the stack for n more entries, where n is the longest path in the acyclic graph resulting from

[2] An SCC of a digraph is a maximal subgraph such that there is a directed path between every pair of its vertices.

```
procedure checkstates();
begin
        for v:=1 to num_of_vertices do
                started[v] := finished[v] := checkset[v] := FALSE;
        for v:=1 to num_of_vertices do
                if started[v]=FALSE then
                        dfs( v )
end; {checkstates}

procedure dfs( v:VERTEX );
begin
        started[v] := TRUE;
        for each edge (v,w) do
                if started[w]=TRUE and finished[w]=FALSE then begin
                        finished[v] := checkset[v] := TRUE;
                        return
                end;
        for each edge (v,w) do
                if finished[w]=FALSE then
                        dfs( w );
        finished[v] := TRUE
end; {dfs}
```

Figure 4.9 : Checking Set Algorithm

removal of the checking states. The approach used here is better for two reasons: (1) a procedure call is required only if there is a potential overflow condition (and parser recovery is desired), and (2) the value compared against the stack pointer is locally optimal. The value `plen` for a checking state can be found by performing a DFS from that checking state, which is halted upon reaching other checking states. Then, `plen` is the maximum depth reached by the DFS. Ideally, the calculation of the `plen` values should occur simultaneously with the checking set algorithm.

Figure 4.10 shows the push graph for the automaton of grammar G_1. The checking set required here is the single vertex number 5, which is the state whose accessing

symbol is "(" in the self-embedding rule $F \rightarrow (E)$. Incidentally, this is a hint that the set of push states for this particular automaton is too large. Indeed, Chapter 5 shows that state 5 is really the only state that requires pushing. Note that Pennello's algorithm applied to this push graph has only a 33% chance of finding the minimum checking set, since vertex 5 is always required, but belongs to an SCC of three vertices. (Vertices that have edges to themselves *must* belong to the checking set.) For the same push graph, the algorithm of Figure 4.9 always discovers the minimum checking set, regardless of how the states are numbered.

A more complicated version of the checking set problem assigns non-unit weights to all the vertices of the push graph, since, in the course of a parse, some of these states are visited more often than others. The expected relative frequencies with which each push state is visited can be estimated by parsing a "representative" set of source files, or by a static analysis of the grammar. Then, the problem is to find a subset of minimum *weight*, rather than minimum cardinality. Of course, the refined problem is

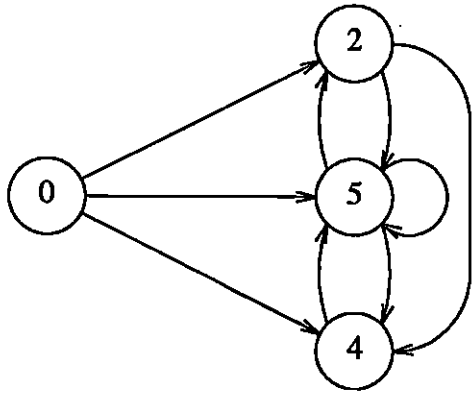


Figure 4.10 : Push Graph of an *MLR* Automaton

even more difficult than the original, and it is unreasonable to expect a parser generator to have access to "representative" source strings.

4.7. Performance of Minpush

The value of the minpush parsing method is measured both statically and dynamically. Rather than present empirical data dependent on local machines, we draw a comparison between minpush *MLALR(1)* parsers and the corresponding *LALR(1)* parsers, for G_1 and the five programming language grammars. Table 4.1 gives static measurements. Both the number of *LR(0)* and *MLR(0)* states for each grammar assume the use of combined shift-reduce actions. For two grammars, there are more *MLR(0)* states than *LR(0)* states due to push-count conflicts. The difference between these numbers gives the number of split states. The column "Conflict Rules" gives the number of grammar rules that have non-unique pop-counts.

The size of an *MLR* parser compared to the corresponding *LR* parser depends on the number of push states relative to the number of *LR(0)* states, the number of split states, and on the number of rules in the minpush parsers that have non-unique pop-counts. From Table 4.1 it is evident that about half of all *MLR(0)* states are push states. Thus, so many push instructions are eliminated that the extra states and extra popping

Grammar	Rules	LR(0) States	MLR(0) States	Push States	Check States	Conflict Rules
G_1	6	8	8	4	1	0
C	217	181	184	99	26	5
Pascal	161	203	203	98	26	0
Oberon	181	171	171	104	18	0
XPL	107	94	98	47	8	2
Algol	92	94	94	51	8	0

Table 4.1 : Static Measurements of Minpush Parsers

code required in the *MLR* parsers for C and XPL are of little significance with respect to space – the *MLR(0)* parsers are smaller.

Dynamic behaviour of minpush parsers is shown in Table 4.2. Four C and four Pascal programs are used as benchmark source sentences. The static ratio of push states to non-push states is more or less mirrored dynamically – about half of all transitions between states are push transitions. The only slow-down in *MLR* parsers comes from the reduction of rules with non-unique pop-counts, when the non-standard extra branch is taken to pop by the non-standard amount. Obviously such reductions do not occur very often, and their effect can be ignored (perhaps up to a dozen extra stack pointer decrements when parsing large programs).

The five test grammars were written for purposes other than this research, and should therefore be reasonably representative. Their only flaw is that they were not written for the purpose of compiling. A grammar edited for use in a compiler generating system is quite different from the test grammars, due to the introduction of extra nonterminals to attach semantic hooks, disambiguate semantic actions, and perhaps help predict syntax errors. Extra nonterminals have a negative effect on minpush parsers because of an increase in the number of origin states. However, these

Grammar	Program	Tokens	Reductions	Shifts	Pushes	Checks	Stack Height MLR	Stack Height LR
C	lzw.c	1324	8661	6438	3567	413	31	40
	gen.c	2423	14873	11221	6071	630	49	64
	items.c	3224	17355	13517	7151	691	40	55
	lalr.c	9101	52134	39798	21468	2328	53	65
Pascal	plane.p	5450	9652	4375	5657	533	62	91
	llpg.p	6224	12044	4488	7464	668	79	113
	llama.p	18782	34851	14900	20905	1790	74	114
	aardvark.p	18509	34701	15083	20598	1738	73	114

Table 4.2 : Dynamic Performance of Minpush Parsers

introduced nonterminals can be "eliminated" using the techniques of the next chapter.

4.8. Comparison with Other Parsers

As we have already seen, a minpush parser behaves very much like an *LR* parser, with the only difference being that the *LR* parser often pushes when the minpush parser does not, and also pops by larger amounts. A minpush parser pushes whenever it enters an origin state, which means that it is pushing in *anticipation* of a rule reduction (*i.e.*, in anticipation of a new reduction path being taken). In light of this property, a minpush parser also shows some similarity to an *LL* parser, which pushes exactly once for every rule that it reduces. The minpush push is, however, a "guess" in that the rule that is subsequently reduced may be any of a number of alternatives, and it is even possible that *no* rule with its origin at the push state is ever reduced. This is possible for any origin state with more than one kernel item, one of which has a shift transition on a terminal symbol. Theorem 4.7 tells us that this can only happen with non-*LL*(1) grammars.

Theorem 4.7: *An MLALR(1) parser created for an LL(1) grammar has exactly one kernel item per parse state.* It is well-known that the *LL(1)* grammars are a subset of the *LALR(1)* grammars [ASU 86]. Related to this is the fact that *LALR(1)* parsers (and *LR(k)* parsers, for that matter) constructed for *LL(1)* grammars have only one kernel item per parse state. Informally, this is because one lookahead symbol is always sufficient to determine the next rule to be reduced, and an *LR* state with more than one kernel item contradicts this assumption (a detailed formal proof can be found in [Tho 77]). Since the *LALR(1)* grammars are contained in the *MLALR(1)* grammars (Theorem 4.5), and every *MLALR(1)* state core is represented at least once in the corresponding *LALR(1)* set of states, the theorem logically follows. □

The next theorem helps to guarantee that a minpush parser need not be much larger than an *LL*(1) parser for a given *LL*(1) grammar.

Theorem 4.8: *An LL(1) grammar does not cause any push-count conflicts in an MLALR(1) adaptively constructed minpush parser.* Earlier in this chapter it is shown that push-count conflict can only occur in states with multiple kernel items. Since there

are no such states in an *MLALR(1)* parser constructed for an *LL*(1) grammar (Theorem 4.7), no push-count conflicts can occur. □

There is a stronger claim to be gleaned from Theorem 4.8. The push-count of an item is only incremented if that item belongs to a state that has at least one closure item. *MLALR(1)* parsers for *LL(1)* grammars have only one kernel item per state, so any closure items are due to the kernel item being of the form $[A \rightarrow \alpha \cdot N \beta]$, where $N \in V_n$. Thus, the pop-count of any reduce item in such a parser must be equal to the number of nonterminals in the right-hand side of the corresponding rule.

Theorem 4.9: *For any given source string derivable from a given LL(1) grammar, the MLALR(1) parser will push no more often, and perhaps less often, than the LL(1) parser.* The *LL*(1) parser pushes exactly once for every nonterminal involved in the derivation of a sentence. The *MLALR*(1) parser pushes when entering a state where at least one rule, but perhaps more than one rule, will subsequently be reduced. The rule that will certainly be reduced has as its LHS the nonterminal at the right of the "dot" in that state's kernel item. If there is any rule of the form $X \rightarrow Y\gamma$ in P, where $Y \in N$, then the state must also contain the closure items $[X \rightarrow \cdot Y\gamma]$ and $[Y \rightarrow \cdot\lambda]$. If the two rules are involved in the leftmost derivation at this point of the parse, the *LL*(1) parser pushes once for each, but the *MLALR*(1) parser only once. □

Despite the fact that the *MLALR(1)* parser pushes no more often, it requires greater stack height, since origin states along a reduction path are not popped until the path's rule is reduced. This also indicates a possible speed benefit. Assuming popping once and popping n times at once takes approximately the same amount of time, the *MLALR* parser saves popping time.

The idea of reducing the number of pushes required in an *LR*-based parser is not new. Purdom and Brown have developed a method for parsing *extended LR* (*ELR*) grammars that pushes only once for each rule reduction [PuB 81]. An *ELR* grammar allows regular expressions to be used for the right-hand sides of rules. Like *MLR* parsers, the Purdom and Brown parsers associate pushing with origin states. The main difference is that the Purdom and Brown parsers push when *leaving* origin states, when

it is better known which rules are on their way to being reduced. Unlike *LL* parsers, it is not important how many rules might be in the process of being reduced. However, their construction method requires kernel items with the same symbols on their right-hand sides to the left of the "dot", which means that new nonterminals sometimes have to be introduced. The effect is similar to the resolution of push-count conflicts by adding special-purpose nonterminals, rather than new states.

CHAPTER 5

DIRECT REDUCTION

5.1. Introduction

The minpush method of Chapter 4 goes a long way towards the elimination of redundant stack use in *LR*-based parsers. However, that method retains at least one push associated with each nonterminal that might be involved in a derivation. Sometimes even this represents redundant pushing. For example, consider a grammar that generates a regular language. Obviously, such a grammar includes at least one nonterminal (the start symbol), and a nontrivial grammar would contain many nonterminals. A minpush parser (and an *LL* parser) constructed for such a grammar pushes at least once for every nonterminal involved in the derivation of a source string, yet, theoretically, a parse of any string generated by the grammar should not require use of a stack at all [HoU 79]. This chapter develops techniques that take advantage of regular grammatical structures to further reduce use of the parse stack. These techniques also eliminate much of the overhead required to make rule reductions in *LR* and *MLR* parsers.

There are three commonly used regular language models: regular (left-linear and right-linear) grammars, nondeterministic and deterministic finite state automata, and regular expressions [ASU 86]. Regular grammars differ from the other two models in that they require the use of nonterminal symbols. Many algorithms exist for the conversion of one model to the other, and for minimizations of each model. In particular, the conversion of a regular grammar to a regular expression involves the elimination of all nonterminals in the grammar [HoU 79]. Similarly, a regular grammar can be converted to a DFA. Note that the nonterminals of a regular grammar are often not required, because only the initial and final states of a DFA are of interest, corresponding to recognition of a string belonging to the regular language. For example, "reductions" in a

scanner are to tokens only; that is, completed regular expressions. No parse trees are required, because it is unnecessary to know how tokens are constructed. Thus, a scanner *recognizes* strings rather than *parses* them. When performing the conversion from a regular grammar to a finite automaton, the reductions of nonterminals which have been eliminated are, in effect, bypassed.

Although the conversion from a regular grammar to a DFA involves the removal of nonterminals, the transformation can often be restricted so that it is possible to determine when the reduction of the eliminated nonterminals would have taken place. Consider the following grammar, G_4:

$$S \rightarrow A\ B\ a\ a \mid A\ A\ a$$
$$A \rightarrow a\ b$$
$$B \rightarrow b \mid B\ d$$

Here, S is the start symbol. A little examination reveals that the grammar generates a regular language. Some factoring and systematic replacement of the nonterminals A and B results in a regular expression for S:

$$\begin{aligned} S &= A\ ((B\ a) \mid A)\ a \\ &= a\ b\ ((b\ d^*\ a) \mid a\ b)\ a \end{aligned}$$

This regular expression can be recognized by the DFA of Figure 5.1. The purpose of the DFA is to merely recognize strings belonging to the regular set generated by S, so all transitions in the DFA are on terminal symbols. However, note that the "reductions" of the rule for A and the two rules for B can be associated with the transitions from state 2 to state 3 ($A \rightarrow a\ b$), state 3 to state 4 ($B \rightarrow b$), and the self-loop on state 4 ($B \rightarrow B\ d$). The reductions are *implicit* whenever these transitions are taken. Since these implicit reductions do not involve any time-consuming overhead such as stack use and indirection, and actually happen on terminal transitions, we can say that the three rules for the two nonterminals A and B are *directly reduced.* In the same vein, the transitions into the final state 8 correspond to the direct reduction of the rules $S \rightarrow A\ B\ a\ a$ and $S \rightarrow A\ A\ a$. In this example, reductions are reported *after* a rule has been recognized, just as in a bottom-up parser.

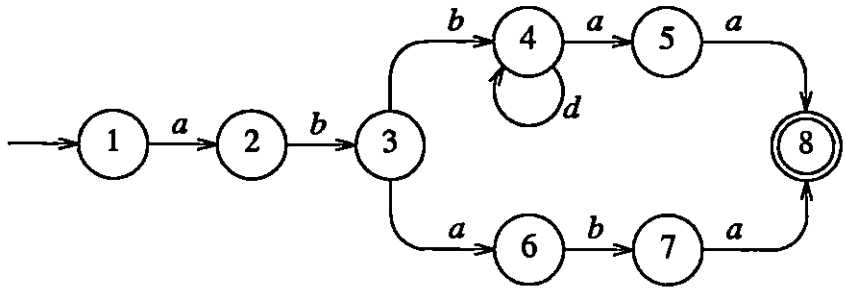


Figure 5.1 : A DFA Recognizer for Grammar G_4

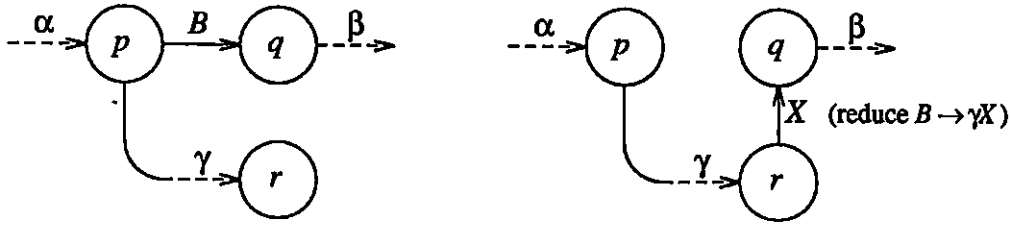
The DFA in Figure 5.1 is not minimal. A minimal automaton would have states 5 and 7 coalesced into one, and there would be only one transition into state 8, on symbol a. However, it would then be impossible to tell which rule for S was being reduced when this transition on a is taken. In general, minimization of a DFA created from a regular grammar does not permit an unambiguous association between individual rules and transitions. However, if we limit the transformation of a regular grammar to a DFA in such a way that it is always possible to associate reductions with transitions, a useful parsing machine results. The next section discusses how this can be done with *LR* automata, created from *LR* grammars and parser generators.

5.2. Local Direct Reduction in LR Automata

The idea of direct reduction can be applied, in a limited sense, to deterministic push-down automata. In particular, the method can be applied to *LR* automata if the power of the parser to recognize rules in the correct order is not affected. This criterion is significant because the correct order of rule reductions for bottom-up parsers is *post-order*, which limits the kind of "regular" rules that can be directly reduced. Direct reduction can be considered a regular optimization, because the efficiency of regular recognizers is exploited.

An *LR* automaton is just a DFA in conjunction with a push-down stack, which is used to find "goto" states when nonterminals are recognized. In terms of tables, the action table of an *LR* parser is equivalent to the transition tables defining a DFA, and the goto table is additional information, accessed via the parse stack, that enables parsing of deterministic context-free languages. However, in conventional *LR* automata the stack is used with all reductions, not just the ones that involve self-embedding or right recursion. This is another example of software over-qualification: a powerful mechanism is being used to do a simple job, resulting in a degradation of efficiency .

In order to use direct reduction with *LR* automata, it is required that the destination (goto) states of nonterminal transitions be accessed from reduce states in a deterministic manner. In general, a transformation as illustrated in Figure 5.2 is required. Figure 5.2 (a) shows how a reduction path in a conventional *LR* automaton might look for the rule $B \rightarrow \gamma X$, and Figure 5.2 (b) shows how reduction of that rule can be bypassed by providing a transition directly from state r to state q. Here, we assume that X is a single grammar symbol, and that the action on symbol X in state r calls for a shift-reduction of the rule $B \rightarrow \gamma X$. In *LR* automata that use combined shift-reduce actions, states like state q in Figure 5.2 do not exist if the action on B in state p is a



(a) Partial *LR* automaton (b) Bypassed reduction

Figure 5.2 : Direct Reduction Transformation

shift-reduction of some rule (then, β would be empty). In cases like this, the transition on X in state r merely inherits the shift-reduce action on B in state p.

Figure 5.2 illustrates the most important condition that must be satisfied for the direct reduction transformation to be possible: state q must be the only goto state (or action) possible if the rule $B \rightarrow \gamma X$ is reduced in state r. If this is the case, direct reduction of the rule $B \rightarrow \gamma X$ from state r is *local*, because there could be other reduction paths for $B \rightarrow \gamma X$ ending in states other than r. They might or might not have the same goto state, q. If there is more than one goto state for a given final item, reduction of that item, and its associated merged reduction paths, is *indirect.* In general, a grammar rule might have several corresponding final items in an *LR* automaton; some of which can be locally directly reduced, and others not.

Direct reduction can be used for both shift-reducible reduction paths (ending in shift-reduce items), and reduction paths ending in inadequate states. There is a subtle difference between these two cases, reflected in the two kinds of final states present in DFAs: those that require "input retraction", and those that do not [HoU 79, ASU 86]. For *LR* parsers using local direct reduction, input retraction means a call to the scanner is not necessary. Thus, if a reduction path ending in a shift-reduce item is directly reduced, a call to the scanner is required at the goto state. If a reduction path ends in a reduce item, no scan call is required at the goto state. It is possible for a state to be the goto state of both reduce and shift-reduce directly reducible items, so some care is required to make sure the scanner is called only if necessary.

The algorithm of Figure 5.3 is used to search through an *LR* automaton for directly reducible final items. The idea is to first assume that all reductions can be made directly, then to weed out those reductions associated with final items that have more than one potential goto action, due to merged reduction paths. This is done by augmenting reduce and shift-reduce items with goto actions, and marking these items as "indirect" as soon as it is discovered that there is more than one possible goto action. After all reduction paths have been examined, those final items that are augmented with only one action mark the end of reduction paths that are locally directly reducible.

Direct reduction increases parsing speed in three ways. First of all, less computation is required to reduce those rules that are directly reduced. A conventional *LR* or

```
procedure getdirect;
begin
        for each state s do
                for each closure item i in s do begin
                        a := goto action of LHS of i;
                        j := reduce item of i;
                        if j is not augmented then
                                augment j with a
                        else if a ≠ goto action of j then
                                mark i as indirect
                end
end; {getdirect}
```

Figure 5.3 : Algorithm to Locate Direct Reduction

MLR parser must pop the stack, access the topmost value, use it to index into the jump table (in hard-coded versions), then finally search a list of nonterminal comparisons in order to branch to the label of the goto state. However, if a rule can be *directly* reduced, the stack is popped (if necessary), and the goto state is found by a direct branch. The indirect accessing, indexing, and searching of normal reduction is bypassed.

The second increase in speed is due to the elimination of redundant nonterminal transitions. If all reduction paths for rules with left-hand side A that originate in state s can be directly reduced, the nonterminal transition on A in state s is redundant, and can therefore be eliminated. Then, indirect reductions that must still access state s, to find the appropriate goto transition, have fewer nonterminal transitions to search through.

The third increase in speed comes from reduced stack use, if direct reduction is used in conjunction with minpush-*LR* parsing. Recall that origin states, defined in Chapter 4, are the states that could be accessed (uncovered) from the parse stack, because they have one or more nonterminal transitions out of them. But direct reduction has the effect of eliminating nonterminal transitions, and some origins lose *all* of

their nonterminal transitions. These states are called *pseudo-origins*, and can be deleted from the set of push states. The decrease in stack use over minpushing alone is as high as 50%, as shown in Section 5.6.

Consider again grammar G_4 from Section 5.1. After augmenting the grammar with the rule $S' \rightarrow \vdash S \dashv$, the reduced *LR(0)* automaton of Figure 5.4 can be constructed. The automaton has seven states, but they are numbered in such a way as to correspond to the DFA constructed for the grammar in Figure 5.1. This is why state 6 is missing. Now, consider the result of the goto state augmenting algorithm. In state 5, the shift-reduce item $[S \rightarrow A\ B\ a \bullet a\ ]$ is augmented with destination state 8, so a direct reduction can be performed there. Similarly, the shift-reduce item in state 7 is augmented with destination 8, so it can be a direct reduction, too. Likewise with state 4, item $[B \rightarrow B \bullet b\ ]$, and destination 4. And finally: state 3, item $[B \rightarrow \bullet b\ ]$, and destination 4. The end result of making these direct reductions and eliminating the nonterminal transitions associated with them is the automaton shown in Figure 5.5. The

I_1: $[S' \rightarrow \vdash \bullet S \dashv]$
$[S \rightarrow \bullet ABaa]$
$[S \rightarrow \bullet AAa]$
$[A \rightarrow \bullet ab]$

I_2: $[A \rightarrow a \bullet b]$

I_3: $[S \rightarrow A \bullet Baa]$
$[S \rightarrow A \bullet Aa]$
$[A \rightarrow \bullet ab]$
$[B \rightarrow \bullet b]$
$[B \rightarrow \bullet Bb]$

I_4: $[S \rightarrow AB \bullet aa]$
$[B \rightarrow B \bullet d]$

I_5: $[S \rightarrow ABa \bullet a]$

I_7: $[S \rightarrow AA \bullet a]$

I_8: $[S' \rightarrow \vdash S \bullet \dashv]$

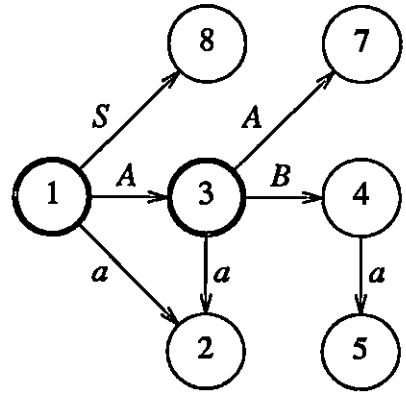


Figure 5.4 : *LR(0)* Sets of Items and Automaton for Grammar G_4

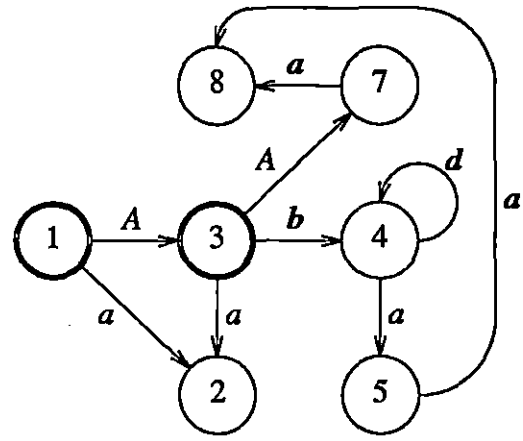


Figure 5.5 : *LR*(0) Automaton for Grammar G_4 with Direct Reduction

transitions on B and S are eliminated. Transitions with associated direct reductions are shown in boldface. Note that no pseudo-origins are created, because of the remaining transitions on A. The parsing automaton is starting to bear a striking resemblance to the DFA of Figure 5.1. Section 5.4 presents a technique that completes the transformation.

It might seem that only unmerged reduction paths are candidates for direct reduction, thus providing the basis for a simpler algorithm than the one shown in Figure 5.3. After all, the previous chapter showed that a reduction path is merged *iff* one or more states in the path, with the exception of the first state, has more than one edge incident upon it. However, it is possible for direct reduction to occur even for merged reduction paths, because distinct origin states does not preclude the possibility of unique destinations. This is shown by the following grammar, G_5, and the reduced *LR(0)* states and automaton shown in Figure 5.6.

$S \rightarrow \vdash A \dashv$
$A \rightarrow a\,B \mid b\,B$
$B \rightarrow C\,b$
$C \rightarrow a\,b$

The grammar generates the regular language $\{ (a \mid b)\, a\, b\, b \}$. There are two merged reduction paths for the rule $C \rightarrow a\,b$, namely 3,6 and 4,6. The rule can be directly reduced, however, as shown by the transition on b from state 6 to state 5. Note that the rule $B \rightarrow C\,b$ *cannot* be directly reduced because stacking is required to choose between the two rules with left-hand side A. However, the two rules for A can be directly reduced by taking the transitions from 3 to 2 or from 4 to 2 when B is recognized.

I_1: $[S \rightarrow \vdash \bullet A \dashv]$
$[A \rightarrow \bullet aB]$
$[A \rightarrow \bullet bB]$

I_2: $[S \rightarrow \vdash A \bullet \dashv]$

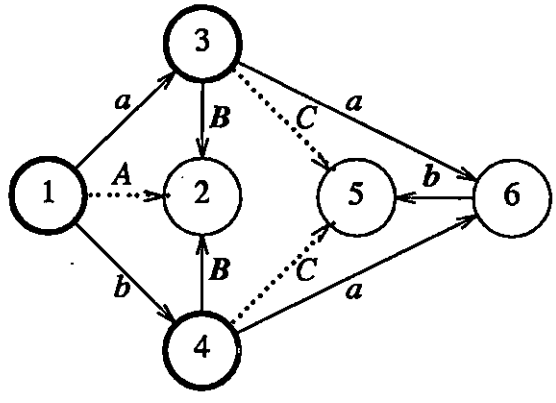
I_3: $[A \rightarrow a \bullet B]$
$[B \rightarrow \bullet Cb]$
$[C \rightarrow \bullet ab]$

I_4: $[A \rightarrow b \bullet B]$
$[B \rightarrow \bullet Cb]$
$[C \rightarrow \bullet ab]$

I_5: $[B \rightarrow C \bullet b]$

I_6: $[C \rightarrow a \bullet b]$

(a) Sets of items for grammar G_5



(b) *LR(0)* automaton showing direct reduction

Figure 5.6 : Merged Direct Reduction Paths

5.3. Trivial Direct Reduction

In *LR* automata there often exist reduction paths of zero length, corresponding to reduction of null productions and shift-reduction of unit productions. Merged reduction paths are clearly impossible in these cases. Thus, they are "trivially" directly reducible because their origin states and their reduce states are identical in each instance.

A unit production is of the form $A \rightarrow X$, where X is a single grammar symbol. In grammars of programming languages, unit productions are often used to specify associativity and precedence of operators within expressions. This use is often strictly syntactic, and unit productions are called *semantically meaningless* if there are no semantic actions associated with their reduction [Jol 79]. A substantial amount of research has been devoted to the "elimination" of these productions from *LR* parsers [AhU 73, KoS 79, LaL 76, Soi 77].

A *unit shift-reduce item* has the form $[A \rightarrow \bullet X]$, where the action on X in the state that contains the item is "shift-reduce by the rule $A \rightarrow X$." Although direct reduction "trivially" applies in these cases, there is a slight complication. The origin state of such a reduction is indeed unique, but *there is no destination state.* The destination state was an *LR(0)* reduce-only state eliminated by using the shift-reduction optimization.

An immediate observation is that the problem of non-existent reduce states is easily solved by simply not using shift-reduction; at least, not with unit productions. However, there is a better alternative that does not sacrifice parsing efficiency, and in fact increases parsing speed. Joliat has defined a simplification of the idea of unit production elimination that applies only to those unit productions that appear as unit shift-reduce items in *LR* automata [Jol 79]. His algorithm is easily applied to hard-coded *LR* parsers, and is, in fact, simply a special case of local direct reduction. If there is no destination state for a particular unit shift-reduce item, the action for that item simply becomes the same action as that for the item's left-hand side. This inheriting of actions is repeated until a goto state is found. (A goto state *must* be found eventually.) Thus, it becomes possible for two or more reductions to be associated with a transition. These reductions are often called *chain reductions*. For example, in Figure 4.3 a shift transition to goto state 7 can be done on *num*, when scanned in states 0 or 5. This

corresponds to direct reduction of the chain $T \rightarrow F \rightarrow num$.

Null productions of the form $A \rightarrow \varepsilon$, where ε is the empty string, occur frequently in grammars of programming languages. One of their uses is for specifying repetition by zero or more times of certain grammatical constructs. For example, a list of identifiers can be expressed by the two rules

$$<list> \rightarrow <list> <id> \mid \varepsilon$$

There are usually no semantic actions associated with the second alternative, $<list> \rightarrow \varepsilon$. The origin states for reduction paths of $<list>$ must each contain the three items $[<list> \rightarrow \bullet <list> <id>]$, $[<list> \rightarrow \bullet \varepsilon]$, and $[A \rightarrow \alpha \bullet <list> \beta]$, where the last item is a kernel item, and β could be empty. Goto states must contain the two items $[A \rightarrow \alpha <list> \bullet \beta]$ and $[<list> \rightarrow <list> \bullet <id>]$. Then, the rule $<list> \rightarrow \varepsilon$ can always be locally directly reduced by providing a direct branch to the appropriate goto state. It is possible that the origin states contain no non-default actions on terminal symbols, in which case the origins are completely redundant, and all transitions into them can go directly to the goto state. The effect is identical to efficient reduction of regular expression rules of the form $A \rightarrow B^*$ in parsers of *extended-LR* grammars [Tho 77, PuB 81].

Another common use of null productions is to provide "semantic hooks" in *LR* grammars [PuB 80]. Unlike null productions used to specify repetition, these productions have semantic code associated with their reduction. In fact, their use is simply one way to have semantic code executed part way through reduction of larger rules. The context of a given semantic hook is usually limited to only one position in one rule, because the required semantic actions are specific to that partial rule reduction. For example, a grammar for Pascal might include the following two rules:

$<stmt> \rightarrow$ while $<\#wmark> <cond>$ do $<compound\text{-}stmt>$;
$<\#wmark> \rightarrow \varepsilon$

Reduction of the semantic hook $<\#wmark>$ might indicate to the compiler that a new label must be generated. It is unlikely that an *LR* machine generated from the grammar has more than one state where $<\#wmark>$ could be reduced. Reduction of $<\#wmark>$

would be a default action in a hard-coded parser, so the required semantic code can occur just prior to the branch instruction that performs local direct reduction of the null rule. Thus, using semantic hooks in hard-coded parsers with direct reduction, semantic code can be inserted at certain non-reduce states, without affecting parsing speed at all.

It is possible for reduction of null productions to be first in a chain of reductions that can be eliminated using the modified Joliat method, described previously in this section. This happens in a state which has an item of the form $[A \rightarrow \bullet B\ ]$, where the action on B is to shift-reduce by the rule $A \rightarrow B$, and also contains the item $[B \rightarrow \bullet \varepsilon]$. Unlike chain reductions consisting entirely of unit reductions, once control is directed to the goto state, there is no need to call the scanner to obtain a new token.

5.4. Direct Reduction by Splitting Paths

In the previous sections of this chapter, direct reduction is applied to existing *LR* or *MLR* automata, as created by a parser generator that does not even consider the idea of direct reduction. If alterations to existing automata are allowed, a great many more cases of local direction can be produced. Chapter 4 subtly introduced the ideas of this section when discussing the resolution of push-count conflict by state splitting, when constructing minpush parsers by adaptation. This is done solely to eliminate all push-count conflicting items from an *LR* automaton in order to create an *MLR* automaton. A frequent side effect is to create direct reduction paths where there were none before. (Although it is not observed with any of the five test grammars, it could theoretically happen that state splitting does *not* create new non-merged reduction paths. For example, consider four merged reduction paths with two conflicting pushcounts, split into two paths of two merged reduction paths each.) As shown earlier in this chapter, unmerged reduction paths are always locally directly reducible. Thus, splitting merged reduction paths can increase parsing speed.

In general, any reduction path merged with one or more other paths can be "split off," state by state, until it is an unmerged path and therefore directly reducible. Splitting starts at the merge state and stops at the final state. The new states inherit all the actions of the original states. The methods of minpush and direct reduction, combined with the splitting of merged reduction paths, allow the automaton for grammar G_4 to

become effectively identical to the DFA shown in Figure 5.1. This final automaton is shown in Figure 5.7, and can be used to parse sentences generated by G_4 without a stack. The split in this case is trivial; the original state 2 becomes state 2 and state 6, thereby removing the nonterminal transitions on A. States 1 and 3 become pseudo-origins, and no push states remain.

It may now appear that since there is an algorithm for making any reduction path an unmerged path, and hence a directly reducible path, any *LR* automaton can be transformed into one that requires no stack. But this is inconsistent with established theory which states that DFAs are not, in general, sufficient to parse context-free languages, because of the self-embedding problem. The seeming paradox is explained in the word "finite" in "deterministic finite automata." It is possible to make every reduction path a direct reduction path, but the result can be an infinite loop, resulting in an infinite automaton. An infinite loop occurs because some merged reduction paths

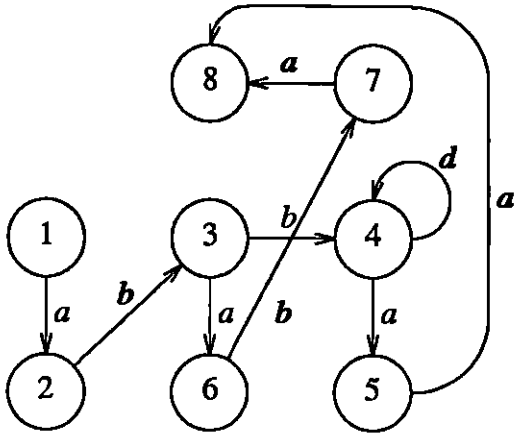


Figure 5.7 : Final *MLR(0)-DR* Automaton for Grammar G_4

cause more merged reduction paths to be created when they are split off. The culprits responsible for this phenomenon are circular reduction paths, which are in turn caused by, not surprisingly, right recursive and self-embedding rules.

Like the adaptive method of minpush parser construction, splitting of non-circular reduction paths to create directly reducible paths is meant to be applied to existing *LR* automata. It is possible to devise an algorithm, incorporating items augmented with both push-counts *and* destination states, that would create an *MLR* parser with direct reduction straight from a grammar. Of course, the algorithm would have to detect instances where the state generation goes into an infinite loop, corresponding to the circular reduction paths in *LR* and *MLR* automata, mentioned above.

5.5. Global Direct Reduction

The use of local direct reduction brings about several complications. Most notably, its use if mostly limited to rules that can be deemed as "semantically meaningless," since it is not always possible to associate semantic actions with local direct reduction transitions, except in special cases. Parser generators that do not know which rules are semantically meaningful cannot afford to "throw away" reductions at will. Also, requiring the user of the generator to somehow declare which rules have associated semantic actions may be too much to ask; this is often an issue of a separate phase of compiler construction. Therefore, it is desirable to apply the idea of direct reduction while working under the constraint that the reduction of all rules must be performed explicitly. In terms of hard-coded parsers, this means that a distinct code address (label) must be associated with each grammar rule. Then, semantic code can be inserted at these addresses, and need not occur more than once for each rule.

In hard-coded *LR* parsers, local direct reduction implicitly completes certain rule reductions by bypassing the reductions. However, if an explicit reduction is required it is still sometimes possible to complete the reduction with a direct branch instruction. Code of the following form is used:

```
r45:  /* semantic code */
      sp -= 3;
      goto st56;
```

In this example, state 56 is *always* the goto state when rule 45 is reduced. Thus, the condition for this form of direct reduction is that for a given rule, there is a unique goto state (or action, in case of shift-reduction). This is called *global direct reduction*, because different reduction paths for a given rule are not considered separately. If there is a unique goto state for a given nonterminal A, all rules with left-hand side A can be globally directly reduced. Therefore, the idea of global direct reduction applies to nonterminal symbols, as well as rules.

The idea of global direct reduction is easily extended by permitting a small number of comparisons to occur just prior to the unconditional branch. These comparisons are between the state uncovered on the stack, and the states whose accessing symbol is the left-hand side of the rule just reduced. Among all states with the same accessing symbol A, the state that has the most incoming transitions is chosen as a default action on A. For example, the reduction block of code for a rule might look as follows:

```
r2:   /* semantic code */
      sp -= 4;
      if (*sp==32) goto st3;
      if (*sp==23) goto st3;
      goto st89;
```

In this case, there are two states in the parser that have the left-hand side of rule 2 as their accessing symbol: state 3 and state 89. State 3 has two transitions entering it, one from state 23 and the other from state 32. All the other transitions on the same symbol lead to state 89, so a branch to this state is the default action.

When one or more non-default comparisons are required, the rule is *pseudo-directly reduced*. If the number of comparisons is kept small, pseudo-direct reduction is still faster than normal reduction. The `lhs` register need not be loaded, since it is not used to find the goto state. If there is another rule with the same left-hand side nonterminal as rule 2, then the last three lines of code in the above example are the same for

the reduction block of that rule. In general, the portion of code that completes global direct reduction for a given nonterminal can be shared among all rules with that nonterminal as their left-hand side. Like state subsumption in Chapter 3, there is sometimes an incidental overhead of one direct branch.

Using global direct reduction has a beneficial effect on stack use in *MLR* parsers. As is the case with local direct reduction, pseudo-origins are created, resulting in fewer states that require pushing, and less maximum stack height. However, use of pseudo-direct reduction does not reduce the number of push states beyond the level attained by local and global direct reduction alone. Even though more states have all their nonterminal transitions "eliminated" through the use of pseudo-direct reduction, such states must still be pushed onto the stack. This is due to the stack access in pseudo-direct reduce blocks, *e.g.*

```
if (*sp==32) goto st3;
```

Thus, there could exist states that have no nonterminal transitions out of them (*i.e.*, no goto blocks associated with them), but must still be pushed.

In hard-coded minpush parsers using global and pseudo-direct reduction, it is necessary to distinguish between four state categories: (1) states with no closure items (non-origins), (2) states with closure items, no nonterminal transitions, and not requiring pushing (pseudo-origins), (3) states with closure items, no nonterminal transitions, but still requiring pushing, and finally (4) states with closure items, nonterminal transitions, and goto blocks. The third and fourth categories comprise the push states. For efficient switching, the states are renumbered so that category four comes first. Then come the states of category three, so that the stack entry size is kept to a minimum. For example, if the number of push states is 256 or less, one byte suffices for each stack entry.

The push-count augmentation algorithm discussed in Chapter 4 must be performed after all direct reduction analysis of an *LR* automaton, because the algorithm must know which states are push states in order to increment item push-counts. State subsumption, discussed in Chapter 3, is dependent on both push-counts and direct reduction, so the proper order in which to construct optimized parsers is to do the direct reduction

analysis, followed by adaptive minpush construction, followed by subsumption analysis. A pleasant side effect of the altered push-counts of *MLR(0)* items is that when we permit one or two comparisons to occur for some rules, (that is, when we use pseudo-direct reduction), push-count conflicts disappear. Thus, *MLR* parsers that use direct reduction sometimes have fewer states and fewer conflict rules than *MLR* parsers that do not use direct reduction. For all five test grammars, *MLALR(1)* parsers using pseudo-direct reduction have the same number of states as the corresponding *LALR(1)* parsers. Also, none of the pseudo-direct parsers have any rules with pop-count conflicts. The suffix *GDR* is used to indicate parsers that use global direct and pseudo-direct reduction.

5.6. Size and Speed of Optimized Parsers

The effects of global direct reduction, alone and in conjunction with minpush parsing, were analyzed by running the parser generation and optimization algorithms on the test grammars. Also, four programs from each of the two largest grammars, C and Pascal, were parsed to obtain experimental measurements. For each grammar, two kinds of parsers were created. One of these is a very large and slow program that keeps track of all actions that it performs, maintaining counts of how many times instructions are executed. Static and dynamic data in Tables 5.1 and 5.2 were found using this program. A variety of directly-executable parsers were generated to gather empirical data, which is given in Table 5.3. *LALR(1)* parsers generated from the same grammars are used as a basis for comparison.

For the results in Tables 5.1 and 5.2, a maximum of two comparisons were allowed to complete pseudo-direct reductions. In Table 5.1, the number of "push" states is the number of states that must be pushed onto the stack because they might subsequently be used to find goto states. The "switch" states are those states which have one or more nonterminal transitions out of them, after nonterminal transitions have been removed through global direct and pseudo-direct reduction. In the "LHS Symbols" section, the first column gives the number of nonterminal symbols which would exist in a conventional *LR* parser (*i.e.* $|V_n|$), the second column gives the number of nonterminals whose rules can all be directly reduced (without any

Grammar	States			LHS Symbols			LHS Comparisons		
	total	push	switch	total	trivial	GDR	direct	switch	max
C	181	81	75	81	43	12	29	175	5
Pascal	203	85	81	53	25	14	19	167	3
Oberon	171	82	50	95	63	7	29	74	2
XPL	97	44	39	49	28	9	14	118	4
Algol	94	46	41	39	23	6	34	117	5

Table 5.1 : Static Measurements of *MLALR(1)-GDR* Parsers

comparisons required), and the third column gives the number of nonterminal symbols remaining in the parsers (*i.e.*, those nonterminals that must be loaded into the `lhs` register just prior to entering the `switch` statement). The total number of comparison instructions required to complete rule reductions in the parsers is given in the first two columns of the "LHS Comparisons" section. The "direct" column refers to pseudo-direct reduction comparisons, and the "switch" column gives the remaining number of true nonterminal comparisons. The "max" column gives the maximum number of nonterminal comparisons required by a goto block for a state.

In Table 5.2, the "switch" reductions are those which are completed via a branch to the `switch` statement to find a nonterminal transition. The "direct" reductions are all the remaining reductions, which are performed either directly or pseudo-directly. In the "Comparisons" section, the "before" column gives the total number of comparison instructions that must be executed in an *LALR(1)* (or *MLALR(1)*) parser without direct reduction, to perform nonterminal transitions. The "switch" column gives the number of comparisons executed in the `switch` statement when direct reduction is used. For both the "before" column and the "switch" column, binary search trees as described in [Pen 86] are used to locate the appropriate nonterminal transition. The figures for using linear search would be far higher.

The total number of comparison instructions performed by the *GDR* parsers to find goto states after rule reductions is given by the sum of the "switch" compares with the "direct" compares. The average number of comparisons performed per rule reduction

Grammar	Program	Reductions		Comparisons			Stack Use	
		switch	direct	before	switch	direct	pushes	height
C	lalr.c	8157	43977	150626	9057	27140	4651	37
	items.c	2773	14582	50503	2989	9034	1550	29
	gen.c	2378	12495	43019	2567	7785	1304	36
	lzw.c	1326	7335	24424	1542	4529	728	23
Pascal	llama.p	17289	17562	46758	16311	18715	10942	64
	aardvark.p	16695	18006	47014	15687	18434	10575	64
	llpg.p	5934	6110	16236	5601	6628	3842	74
	plane.p	4596	4656	12995	4682	5233	2988	52

Table 5.2 : Dynamic Performance of *MLALR(1)-GDR* Parsers

is simply the quotient of this sum and the total number of reductions, which gives about 1.0 for Pascal and 0.7 for C. The "pushes" column gives the number of push instructions executed – note that the values are substantially smaller than those in Table 4.2, especially for C programs. Also, the maximum stack height reached is smaller.

Table 5.3 gives empirical data which were gathered from the two largest grammars, and the largest test program for each. The computer used for these experiments is a Sun 3/280S running UNIX 4.2. Parsers were compiled using the optimizer (-O flag). The C program, *lalr.c*, is an *LALR(1)* parser generator, and the Pascal program, *llama.p*,

Grammar	Parser size (bytes)					
C	25724	16340	14176	11892	6952	6848
Pascal	9720	7340	6312	5480	5560	5580
Program	Parsing speed (in seconds)					
lalr.c	0.33	0.34	0.32	0.22	0.11	0.11
llama.p	0.20	0.20	0.18	0.16	0.15	0.13
	LALR(1)	Subsumption	MLALR(1)	Max=0	Max=1	Max=2

Table 5.3 : Absolute Size and Speed of Hard-Coded Parsers

is a scanner generator. They occupy about 33000 and 98000 bytes of file space, respectively. The parsers do not have any syntax error correction ability, and the given sizes do not include 200 bytes and 100 bytes allocated for the parse stack in the *LALR(1)* and *MLALR(1)* versions, respectively. The given parse times do not include the scanning time. Parse times were calculated by running the scanner and parser together on the test programs, then running the scanner alone, and averaging over ten test runs. The difference between these two times is the parse time.

The first column refers to hard-coded *LALR(1)* parsers that do not use any of the techniques described in Chapters 3, 4 and 5 in this thesis. The second column gives the results when state subsumption, described in Chapter 3, is applied to the parsers. All the remaining columns include the optimization of state subsumption to reduce space requirements. The next column improves upon the previous by using the minpush parsing algorithm of Chapter 4. The last three columns give the results if global direct reduction and pseudo-direct reduction, from Chapter 5, are used in conjunction with minpushing. In the column "Max=0," no pseudo-direct reductions were permitted. In the columns "Max=1" and "Max=2," up to one or two comparisons were allowed to complete pseudo-direct reductions. Parser sizes and execution times increase when allowing more than two comparisons for pseudo-direct reductions, so two is the recommended number for this parameter.

The naïve implementations are fast already, but the optimizations of minpushing and global direct reduction decrease execution times by a factor of about 66% for C and 35% for Pascal. In addition, parser sizes drop by about 74% for C and 43% for Pascal, resulting in directly-executable parsers that are small enough to be used in modest computing environments. The main reason for the dramatic improvement in the size of the C parsers is that so many parse states have very many non-default terminal transitions (one state has thirty!), and most of these belong to equivalence classes or can be subsumed.

CHAPTER 6

CONCLUSIONS

6.1. Summary

Mechanically generated directly-executable parsers are useful in any computer application that requires a parser, and is run in a computing environment that can provide enough memory to store the parser. If properly designed, directly-executable parsers run much faster than their table-driven counterparts. This thesis is concerned with the optimization of directly-executable *LR* parsers – optimizations that make the parsers both smaller and faster than naïvely constructed ones. The C programming language was chosen for implementation of the constructor algorithms and for the generated hard-coded parsers, thus ensuring a high degree of portability.

Certain desirable properties of *LR* parsers should not be compromised when modifying them. The optimizations developed in this thesis are constrained to satisy three basic criteria:

(1) The language accepted by the *LR* parser and the modified, optimized *LR* parser should be exactly the same.

(2) The ability of an *LR* parser to detect syntax errors before the first invalid token is shifted should not be changed.

(3) The modified parser should be able to contain semantic code, either in-line or via procedure call, at any point in a production rule where an *LR* parser could contain the same semantic action.

In addition to satisfying the basic criteria, the optimization algorithms are invisible to users of the parser generator.

Some optimizations already in widespread use with table-driven *LR* parsers are immediately applicable to directly-executable *LR* parsers. In particular, combined shift-reduce actions can be used, and shift-reduce transitions caused by semantically insignificant unit productions are easily eliminated. Disregarding implementation details, three main categories can be identified for further optimization of directly-executable *LR* parsers. They are (1) terminal transitions, (2) nonterminal transitions, and (3) use of the parse stack.

Terminal actions can be shared among many states of a parser by considering equivalence classes of states, nearly-equal states, and subsumable states. This is worthwhile for those states with many terminal transitions, because the occasional overhead of one additional branch instruction becomes insignificant. A state must have a certain minimum number of terminal transitions in order to be considered for state subsumption; a value of approximately seven is recommended. State subsumption is a space optimization only, and about 40% of all terminal transitions in a directly-executable *LR* parser can be eliminated using chain decompositions of subsumption graphs. The parser for the C language, which has a very "dense" grammar, is smaller by approximately 9000 bytes, with no measurable speed penalty.

Nonterminal transitions that complete rule reductions are the main cause of slow operation in *LR* parsers, when compared to *LL* parsers. In directly-executable *LR* parsers, nonterminal transitions can be accomplished by searching for state number within nonterminal symbol, or by searching for nonterminal symbol within state number. The latter technique requires use of the "switch" statement in C to perform indirect branching, but the former technique, called pseudo-direct reduction, does not require this extra overhead. It turns out that a mixed approach is best, resulting in search lists that are far shorter than in a pure approach. More than 60% of all nonterminal transitions in an *LR* parser can be eliminated, when permitting a maximum of only two comparisons for state numbers within rule reductions. Direct reduction of semantically insignificant productions further reduces the number of nonterminal transitions, and also eliminates redundant branching.

From a theoretical viewpoint, the main contribution of this thesis is the *minpush-LR*, or *MLR*, parsing and parser contruction algorithms presented in Chapter 4.

Minpush parsing is based on *LR* parsing, but is also closely related to *LL* parsing. In short, the difference between an *LR* parser and a corresponding *MLR* parser is that the former pushes all states onto the parse stack, but the latter pushes only those states with nonterminal transitions. In *LALR(1)* automata constructed from grammars of programming languages, these "origin" states account for about 60% of all parser states, assuming *LR(0)* reduce states have already been eliminated. Experiments on large sample programs indicate that from about 40% to 50% of all shift transitions do not require pushing. When minpush parsing is augmented with direct reduction techniques, the number of states that require pushing is further diminished by approximately 20% of the origin states. A *minpush-LALR(1)* (or *MLALR(1)*) parser generator can be created by some simple modifications to an *LALR(1)* parser generator.

It is important that the optimization algorithms are implemented in correct order, due to their interdependence. Minpush parsing depends upon direct reduction, and state subsumption depends upon both local direct reduction and minpushing. Therefore, optimization proceeds by construction of an automaton from *LR* parse tables, followed by local and global direct reduction analysis. Then, the automaton is converted to a minpush automaton by augmenting items with push-counts, and splitting states where necessary. Finally, equivalence classes of states and subsumption chains are found. At this point, the automaton can be converted to a directly-executable parser.

The results presented in various tables throughout the thesis were obtained with a large generator/simulator which was developed over the past year. Faster parser operation is proven by analyses of how many instructions of various types must be executed to complete a parse. Empirical measurements confirm analysis – for example, parsing times for C programs drop by about two-thirds when using the optimizations. However, the speed of a directly-executable parser written in C is dependent on the quality of the C compiler, code optimizer, and the machine architecture in a given computing environment.

6.2. Undeveloped Areas

Although all parser optimizations discussed in this thesis guarantee that the parsers do not shift any invalid tokens, little attention has been given to the issue of recovery

from syntax errors. This area is of great importance to compiler development. The *LR* and *MLR* parsing methods are similar in that the parse stack of the latter can be converted to the parse stack of the former. Thus, standard *LR* error-recovery algorithms can be used. In addition, the parse stack of an *MLR* parser is similar to the parse stack of an *LL* parser, so that error recovery algorithms used for *LL* parsers might also be usable in *MLR* parsers after some modification. In any case, work is required in this area before *MLR* parsers, with or without direct reduction and state subsumption, can be used in practical applications.

The emphasis in this research has been on efficient parsing algorithms, rather than efficient parser generation algorithms. Thus, little or no analysis has been done on the time and space complexity of the generation algorithms. Some examples are the checking state algorithm, the push-count augmentation algorithm, the state subsumption algorithms, and the methods used to determine the nonterminals whose rules can be directly reduced. These algorithms are based primarily on graph theory, and therefore require the attention of this branch of computing science.

6.3. Future Research

The methods of Chapter 4 and Chapter 5 succeed in eliminating many push instructions from directly-executable *LR* parsers. However, the methods do not guarantee *minimum* pushing by any means. Whether or not a truly minimum machine is attainable is an interesting question. A possible research project is to analyze *LR* automata to obtain a parser that pushes only as much as it needs to, and then allow finite state splitting to further reduce the number of push states. Pushes could become attributes of edges (transitions) rather than vertices (states). A good starting point is to consider pushing on transitions leaving origin states, rather than those that enter origin states. This has the desirable side effect of reducing nonterminal search, as well.

The main idea behind the algorithms developed in this thesis is that a source sentence should be parsed with as little effort as possible. This is called "adaptive" parsing. Attention is confined to *LR*-based parsers, but the same idea should be applied to larger grammar classes. For example, the method of Earley [Ear 70] can be used to parse any unambiguous context-free language, but it runs very slowly. It might be

possible to simplify an Earley parser so that if the grammar to be recognized is simple, or has simple parts, then these parts can be parsed much faster than the worst-case parts. Similarly, the greater part of a context-sensitive grammar might be parsable using *LR*, *LL*, or even regular techniques. Research in this area would have immediate application to natural language understanding systems.

REFERENCES

[AhU 72] Aho, A.V. and J.D. Ullman, "Optimization of *LR(k)* parsers," *Journal of Computer and Systems Sciences* **6:6**, June 1972, pp. 573-602.

[AhU 73] Aho, A.V. and J.D. Ullman, "A Technique for Speeding Up *LR(k)* Parsers," *SIAM Journal of Computing* **2:2**, June 1973, pp. 106-127.

[AhJ 74] Aho, A.V. and S.C. Johnson, "*LR* Parsing," *Computing Surveys* **6:2**, June 1974, pp. 99-124.

[AJU 75] Aho, A.V., S.C. Johnson and J.D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Communications of the ACM* **18:8**, Aug. 1975, pp. 441-452.

[ASU 86] Aho, A.V., R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[AEH 73] Anderson, T., J. Eve and J.J. Horning, "Efficient *LR*(1) Parsers," *Acta Informatica* **2:1**, 1973, pp. 12-39.

[Bac 76] Backhouse, R.C., "An Alternative Approach to the Improvement of *LR(k)* Parsers," *Acta Informatica* **6:3**, 1976, pp. 277-296.

[BaE 76] Bauer, F.L. and J. Eickel (Eds.), *Compiler Construction - An Advanced Course*, Lecture Notes in Computer Science **21**, Springer-Verlag, 1976.

[CoR 78] Cohen, J. and M.S. Roth, "Analyses of Deterministic Parsing Algorithms," *Communications of the ACM* **21:6**, June 1978, pp. 448-458.

[DDH 84] Dencker, P., K. Durre and J. Heuft, "Optimization of Parser Tables for Portable Compilers," *ACM TOPLAS* **6:4**, Oct. 1984, pp. 546-572.

[DeR 71] DeRemer, F.L., "Simple *LR(k)* Grammars," *Communications of the ACM* **14:7**, July 1971, pp. 453-460.

[DeP 79] DeRemer, F.L. and T. Pennello, "Efficient Computation of *LALR*(1) Look-Ahead Sets," *SIGPLAN Notices* **14:8**, Aug. 1979, pp. 176-187.

[Ear 70] Earley, J., "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM* **13:2**, Feb. 1970, pp. 94-102.

[FiL 88] Fischer, C.N. and R.J. LeBlanc, Jr., *Crafting a Compiler*, Benjamin/Cummings, 1988.

[GaJ 79] Garey, M.R. and D.S. Johnson, *Computers And Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979.

[Ger 87] Gerardy, R., "Experimental Comparison of Some Parsing Methods," *SIGPLAN Notices* **22:8**, Aug. 1987, pp. 79-88.

[GHJ 79] Graham, S.L., C.B. Haley and W.N. Joy, "Practical *LR* Error Recovery," *SIGPLAN Notices* **14:8**, Aug. 1979, pp. 168-175.

[Gra 87] Gray, R.W., "Automatic Error Recovery in a Fast Parser," *Conference Proceedings, USENIX TCE*, June 1987, pp. 337-346.

[HaS 84] Harbison S.P. and G.L. Steele, *C - A Reference Manual*, Prentice-Hall, 1984.

[Hoa 81] Hoare, C.A.R., "The Emperor's Old Clothes," *Communications of the ACM* **24:2**, Feb. 1981, pp. 75-83.

[HoU 79] Hopcroft, J.E. and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[Hor 86] Horspool, R.N., *C Programming in the Berkeley UNIX Environment*, Prentice-Hall, 1986.

[HoL 87] Horspool, R.N. and M.R. Levy, "*Mkscan* - An Interactive Scanner Generator," *Software - Practice and Experience* **17:6**, June 1987, pp. 369-378.

[HoL 88] Horspool, R.N. and M.R. Levy, "MkTypeCheck," unpublished manuscript, Dept. of Computer Science, University of Victoria, 1988.

[Hun 85] Hunter, R., *COMPILERS: Their Design and Construction Using Pascal*, John Wiley, 1985.

[IcM 70] Ichbiah, J.D. and S.P. Morse, "A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars," *Communications of the ACM* **13:8**, 1970, pp. 501-508.

[Joh 79] Johnson, S.C., "YACC - Yet Another Compiler Compiler," *UNIX Programmer's Manual*, 7th Ed., Vol. **2B**, Jan. 1979.

[Jol 74] Joliat, M.L., "Practical Minimization of *LR(k)* Parser Tables," *Proc. IFIP Congress*, Amsterdam, Netherlands: North-Holland, 1974, pp. 376-380.

[Jol 76] Joliat, M.L., "A Simple Technique for Partial Elimination of Unit Productions from *LR(k)* Parser Tables," *IEEE Trans. on Computers* **C-25:7**, 1976, pp. 763-764.

[KeR 78] Kernighan, B.W. and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[Knu 65] Knuth, D.E., "On the Translation of Languages from Left to Right," *Information and Control* **8:6**, 1965, pp. 607-639.

[Knu 71] Knuth, D.E., "Top-Down Syntax Analysis," *Acta Informatica* **1:2**, 1971, pp. 79-110.

[Kor 69] Korenjak, A.J., "A Practical Method for Constructing *LR(k)* Processors," *Communications of the ACM* **12:11**, Nov. 1969, pp. 613-623.

[KoS 79] Koskimies, K. and E. Soisalon-Soininen, "On a Method for Optimizing *LR* Parsers," *International Journal of Computer Mathematics* **A:7**, 1979, pp. 287-295.

[LLH 72] LaLonde, W.R., E.S. Lee and J.J. Horning, "An *LALR(k)* Parser Generator," *Information Processing* **71**, North-Holland, 1972, pp. 513-518.

[LaL 76] LaLonde, W.R., "On Directly Constructing *LR(k)* Parsers Without Chain Reductions," *Proc. 3rd ACM Symposium on Principles of Programming Languages*, 1976, pp. 127-133.

[Mac 86] Machanick, P., "Are *LR* Parsers Too Powerful?," *SIGPLAN Notices* **21:6**, June 1986, pp. 35-40.

[Meh 84] Mehlhorn, K., *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, 1984.

[MiS 73] Mickunas, M.D. and V.B. Schneider, "A Parser-Generating System for Constructing Compressed Compilers," *Communications of the ACM* **16:11**, Nov. 1973, pp. 669-676.

[Pag 72] Pager, D., "On Eliminating Unit Productions from *LR(k)* Parsers," Automata, Languages, and Programming (J. Loeckx, ed.), 2nd Colloquium, *Lecture Notes in Computer Science* **14**, Springer 1974, pp. 242-254.

[Pag 77a] Pager, D., "A Practical General Method for Constructing *LR(k)* Parsers," *Acta Informatica* **7:3**, 1977, pp. 249-268.

[Pag 77b] Pager, D., "Eliminating Unit Productions from *LR* Parsers," *Acta Informatica* **9:1**, 1977, pp. 31-59.

[Pen 86] Pennello, T.J., "Very Fast *LR* Parsing," *SIGPLAN Notices* **21:7**, July 1986, pp. 145-151.

[Pur 74] Purdom, P.W., "The Size of *LALR*(1) Parsers," *BIT* **14:3**, July 1974, pp. 326-337.

[PuB 80] Purdom, P.W. and C.A. Brown, "Semantic Routines and *LR(k)* Parsers," *Acta Informatica* **14:4**, 1980, pp. 299-315.

[PuB 81] Purdom, P.W. and C.A. Brown, "Parsing Extended *LR(k)* Grammars," *Acta Informatica* **15:2**, 1981, pp. 115-127.

[Röh 80] Röhrich, J., "Methods for the Automatic Construction of Error Correcting Parsers," *Acta Informatica* **13:2**, 1980, pp. 115-139.

[Soi 77] Soisalon-Soininen, E., "Elimination of Single Productions from *LR* Parsers in Conjunction with the Use of Default Reductions," *Proc. 4th ACM Symposium on Principles of Programming Languages*, 1977, pp. 183-193.

[TaY 79] Tarjan, R.E. and A.C. Yao, "Storing a Sparse Table," *Communications of the ACM* **22:11**, 1979, pp. 606-611.

[Tho 77] Thompson, D.H., "The Design and Implementation of an Advanced *LALR* Parse Table Constructor," *Computer Systems Research Group*, Technical

Report **CSRG-79**, University of Toronto, Apr. 1977.

[TrS 85] Tremblay, J.-P. and P.G. Sorensen, *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985.

[WaC 85] Waite, W.M. and L.R. Carter, "The Cost of a Generated Parser," *Software - Practice and Experience* **15:3**, March 1985, pp. 221-237.

[WaG 84] Waite, W.M. and G. Goos, *Compiler Construction*, Springer-Verlag, 1984.

[WiW 66] Wirth, N. and H. Weber, "EULER - A Generalization of ALGOL and its Formal Definition: Parts I and II," *Communications of the ACM* **9:1,2**, 1966, pp. 13-25,89-99.

[Wir 85] Wirth, N., "From Programming Language Design to Computer Construction," *Communications of the ACM* **28:2**, Feb. 1985, pp. 159-164.

VITA

Surname: WHITNEY **Given Names:** MICHAEL JAMES

Place of Birth: Edmonton, Alberta, Canada **Date of Birth:** June 12, 1960

Educational Institutions Attended, with Dates of Entering and Leaving:

University of Alberta, Edmonton	1982 to 1986
University of Victoria, B.C.	1986 to 1988

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B.Sc.	University of Alberta, Edmonton	1986

Honours and Awards:

Dome Petroleum Ltd. Scholarship	1984-1985
MacDonald-Dettwiler Ltd. Scholarship	1985-1986
NSERC Postgraduate Scholarship	1986-1987 1987-1988

Publications:

(1) **Decisive Differences and Partial Differences for Fault Detection in MVL Circuits**, (with Jon Muzio), *Proceedings of the Eighteenth IEEE International Symposium on Multi-valued Logic*, May, 1988.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make *single copies only* for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Optimization of Directly Executable LR Parsers

Author [redacted]

MICHAEL JAMES WHITNEY

Aug. 2, 1988

Date