Faster Tomita Parsing

by

John Daniel Aycock
B Sc , University of Calgary, 1993

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming

[redacted]

Dr R N Horspool, Supervisor (Department of Computer Science)

[redacted]

Dr M R Levy, Departmental Member (Department of Computer Science)

[redacted]

Dr H A Muller, Departmental Member (Department of Computer Science)

[redacted]

Dr N J Dimopoulos, External Examiner
(Department of Electrical and Computer Engineering)

Abstract

Tomita's parsing method, or generalized LR parsing, was designed to parse ambiguous grammars efficiently Tomita uses specific linear-time LR parsing techniques as long as possible, falling back on more expensive general techniques when necessary

Much research has addressed speeding up LR parsers, in this thesis, we argue that this previous work is not transferable to Tomita parsers To speed up LR parsers, we reduce LR parsing overhead two ways grammar transformations unroll recursion, and larger finite automata in the parser trade space for time

We have devised a variant of Tomita's algorithm which incorporates our low-overhead LR parsers Our timings show that our Tomita variant gives an order of magnitude improvement for the worst case ambiguous grammar on most inputs, several orders of magnitude improvement are seen on larger grammars

Examiners

[redacted]

Dr R N Horspool, Supervisor (Department of Computer Science)

[redacted]

Dr M R, Levy, Departmental Member (Department of Computer Science)

[redacted]

Dr H A Muller, Departmental Member (Department of Computer Science)

[redacted]

Dr N J Dimopoulos, External Examiner
(Department of Electrical and Computer Engineering)

Table of Contents

List of Tables

List of Figures

Acknowledgments

First, many thanks to Dr. Horspool, who suggested this topic to begin with. He supplied much useful feedback, support, and constructive criticism.

I would also like to thank Dr. Rokne from the Department of Computer Science at the University of Calgary. When I was stuck for computing horsepower, he let me have access to their enormous Silicon Graphics research machine.

Mike Zastre listened to my innumerable tales of woe as we made the daily pilgrimage to Tim Horton's. Shannon Jaeger and Jim Uhl did an excellent job proofreading. Now if only I weren't too stubborn to listen to them!

This work could not have been completed without the constant support of my family. Melissa, who wasn't old enough to understand why Daddy went away to "work" so much, Shannon, who did understand and let me go anyway: thank you both.

Chapter 1

Introduction

She saw the boy with the telescope

Does the boy have the telescope, or not? The answer lies in the structure of the sentence, and whether "with the telescope" modifies "saw" or "boy." The process of determining the structure of an input — like the above sentence — is called parsing. A computer program which parses input is called a parser.

Parsing is a fundamental topic in computer science because it has so many application areas. Programming language compilers, natural languages, database query languages, and document markup languages like HTML all require parsers, to name but a few examples.

To guide a parser in determining an input's structure, it has a set of rules which

describe all possible valid inputs. This set of rules is referred to as a grammar. Unfortunately, grammars are not always able to describe a unique, unambiguous way to interpret every input. As with the English grammar that imposes a structure on the above sentence, some grammars are ambiguous.

Like humans, parsers have an easier time understanding input which has only a single interpretation. More importantly, parsers which deal strictly with unambiguous grammars can operate much faster than parsers for ambiguous grammars. This is crucial when one considers that the speed of input recognition is highly visible to users. As a result, most artificial languages (such as those for programming languages) have unambiguous grammars by design, and much research has addressed speeding up parsers for unambiguous grammars. However, applications like natural language understanding are rarely able to choose a convenient grammar, so there is still a need for fast parsers for ambiguous grammars.

In his Ph.D. thesis [38], Tomita presented a method for parsing which combined the best of both worlds. A Tomita parser employs techniques used for unambiguous grammars as long as possible, then falls back on more expensive techniques to handle ambiguity when necessary. Besides natural languages, Tomita parsers are well suited to parse any grammar which contains areas of ambiguity.

Our research has focused on speeding up Tomita parsers. Returning to first princi-

ples, we present a different way to construct these parsers, and show that our method results in faster Tomita parsers.

This thesis is arranged in the following manner. We introduce grammars, languages, and parsing formally in Chapter 2. Chapter 3 is a review of LR parsers, the specific class of parsers that we use in our work. Chapter 4 describes our method for building faster LR parsers. In Chapter 5, we apply that method to a Tomita parser, a parsing technique which can efficiently handle ambiguity. Chapter 6 discusses implementation details. Finally, Chapter 7 presents avenues for future work and concludes the thesis.

Chapter 2

Languages, Grammars, and Parsing

Since parsing has a large number of diverse applications, it is not surprising that there is a large body of theoretical work to support it. In this chapter, we present some of this material as it relates to our work.

2.1 Languages

Formally, a language is a set of strings over an alphabet [29]. An alphabet is a finite set of symbols, and a string is any finite sequence of alphabet symbols. A language itself

may be either finite or infinite. For example, take the alphabet $\Sigma = \{a, b\}$. Then a, aba, and $abbbbb$ would all be examples of strings, the languages over Σ include $\{a, ab, ba\}$ and Σ^*. By Σ^* we mean the Kleene closure of Σ, the set formed through concatenation of zero or more strings in Σ [29].

Using the notation of [1], the symbol ϵ denotes an empty string with no symbols, and \$ is a symbol that acts as a end-of-input sentinel at the end of strings. Lowercase letters late in the English alphabet, such as w, are used to represent arbitrary strings.

Chomsky [14, 12] classified languages into four types. Of those language classes, only two are known to be recognized by efficient parsing methods, so we restrict our attention to them: regular languages and context-free languages (CFLs). Fortunately, many "interesting" languages for practical purposes, such as programming languages, belong to these two classes. These classes of languages may be distinguished by one of the means used to describe them — grammars.

2.2 Grammars

A grammar consists of a finite set of rules. By applying these rules, a grammar actually behaves as a generative device for a language, and is able to produce all strings that belong to it [12]. So to parse an input string and divine its structure, a

parser must operate backwards in the sense that it needs to determine which rules were used, and in what order, to create a particular input. A parser must also be able to detect invalid inputs that are not part of the language defined by the grammar.

Grammar rules themselves are comprised of two different types of symbols:

1. Terminal symbols. These are just symbols from a language's alphabet, and will be written using lowercase English letters like b [1]. Depending on the particular grammar, other symbols like parentheses may be used as terminal symbols for clarity.

2. Nonterminal symbols. These are symbols may be thought of as "variables" that may be substituted with sequences of terminals and nonterminals [42]. The grammar's rules define all valid substitutions. Uppercase alphabetic letters early in the alphabet such as B are used to represent nonterminals [1].

Lowercase Greek letters represent strings of terminal and nonterminal symbols, including the empty string, uppercase English letters late in the alphabet (e.g., X) stand for a single terminal or nonterminal [1].

Returning to the analogy of nonterminals as variables, a grammar rule $A \to \alpha$ indicates that the nonterminal A may be substituted with α wherever A appears. When this happens, A is said to derive α, written $A \Longrightarrow \alpha$. Repeated derivations

may occur, $A \stackrel{*}{\Longrightarrow} \alpha$ means that A derives α in zero or more steps. If a situation arises where there are several nonterminals that could be substituted for, substituting the rightmost nonterminal yields a rightmost derivation, denoted $A \stackrel{*}{\underset{rm}{\Longrightarrow}} \alpha$.

In the grammar of Figure 2.1, we have the terminals $\{a, b, c, d\}$ and the nonterminals $\{S, A, B\}$. The language defined by this grammar is the finite set $\{c, acdb\}$. Some of the statements we can make about this grammar are

$$S \underset{rm}{\Longrightarrow} aABb \underset{rm}{\Longrightarrow} aAdb \underset{rm}{\Longrightarrow} acdb$$

$$S \stackrel{*}{\underset{rm}{\Longrightarrow}} acdb$$

$$S \stackrel{*}{\Longrightarrow} acdb$$

Notice that our derivations above all began with S. Grammars have a distinguished start symbol from which derivations begin — conventionally, the start symbol is the nonterminal on the left-hand side of the first grammar rule. In practice, grammars will sometimes be augmented with a new start symbol S' and a rule $S' \rightarrow S\ \$$. Using an augmented grammar simplifies specification and implementation of some algorithms [10].

Now let us be more precise. A grammar is a four-tuple $G = (N, \Sigma, R, S)$, where

N is a finite set of nonterminal symbols,

Σ is a finite set of terminal symbols, $\Sigma \cap N = \emptyset$, and

S → a A B b
S → c
A → c
B → d

Figure 2.1 A simple grammar

$S \in N$ is a start symbol [2, 29]

R is a set of rules whose composition depends on the type of grammar being defined. Remember that we are only interested in regular languages and context-free languages, and that classes of languages are defined by classes of grammars. Regular languages are defined by regular grammars, for whom R is a finite subset of $N \times (\Sigma \cup (\Sigma \times N))$[1]. Context-free grammars (CFGs) define context-free languages, here R is a finite subset of $N \times (N \cup \Sigma)^*$. For example, the grammar used in Figure 2.1 was a CFG.

The language defined by a grammar G is denoted L(G). G is ambiguous if and only if two or more distinct rightmost derivations exist for a given input string.

[1]There are other equivalent definitions, but this one is convenient for discussion purposes.

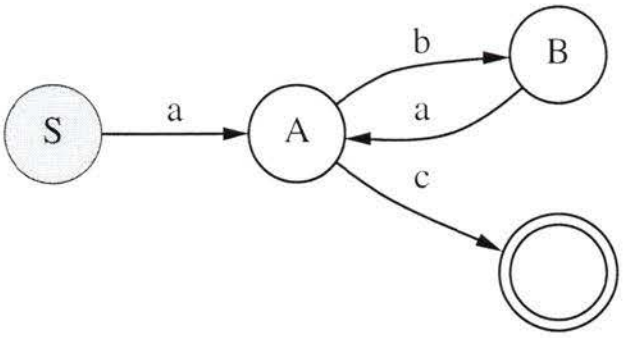
2.3 Parsing

Now that we have defined what we mean by languages and grammars, we can talk about the machinery used in parsing. As we implied, the job of a parser is to check the validity of an input string according to a grammar. It does so by determining a sequence of derivations from the grammar's start symbol that would result in the production of the input string. Different types of parser go about this task in different ways, and the method a parser uses determines the class of languages it can recognize, some classes are larger than others. Given this fact, we treat the parsing of regular grammars and CFGs separately.

2.3.1 Parsing Regular Grammars

To parse regular grammars, one makes use of the fact that regular grammars are equivalent in expressive power to finite automata [14]. A finite automaton (FA) is a state machine. It is comprised of a finite set of states, and the transitions between them. Beginning in a unique start state, a FA will make one or more transitions between states for every input symbol it reads. At the end of the input, the FA accepts the input as valid if the FA is in one of a set of final states.

A direct conversion is possible from a regular grammar to a FA [5, 7].



S → a A
A → b B
A → c
B → a A

Figure 2.2 Regular grammar and associated FA

1. Create a FA state for every nonterminal in the grammar, plus a final state F.
2. For every grammar rule A → a B, add a directed edge from state A to state B labeled a.
3. For every rule A → a, add an edge from state A to state F.
4. The state corresponding to the grammar's start symbol is the start state.

To illustrate, Figure 2.2 shows a regular grammar and its associated FA. FA states are drawn as circles, the shaded circle indicates the start state, and the double-circle is a final state.

The set of regular languages is a proper subset of the set of CFLs [14]. Why would one not eschew parsing techniques for regular grammars in favor of more powerful

ones used for parsing CFGs? While this would certainly be possible, it turns out that finite automata have much lower overhead than the corresponding parsers for CFGs using a CFG parser for anything but CFGs would be overkill

2 3.2 Parsing Context-Free Grammars

How then can CFGs be parsed? Unfortunately there are only a few general methods known, such as Earley's algorithm [9, 31] Even the Tomita algorithm we use in this thesis is unable to handle certain ambiguous grammars [12] In any case, general CFG parsing methods tend to have high overhead, and can have poor worst-case performance $O(n^3)$ for Earley on ambiguous grammars [9], and Tomita can be exponentially slower than Earley [19] (On unambiguous grammars, Earley has an $O(n^2)$ worst case) In practice, often parsers are used for CFGs which accept only a subset of CFLs

One of the more important CFL subsets is the LR class introduced by Knuth [24] The LR class consists of those CFLs whose grammars may be recognized by LR parsers, these grammars are unambiguous by definition Not only are many grammars — like those for programming languages — recognizable using LR parsers, but deterministic parsing methods are known which execute in linear time [1] Because of its importance, and because Tomita parsing relies upon it, LR parsing is the topic

of the next chapter

Chapter 3

Review of LR Parsing

By definition, a LR parser reads its input from left to right, and produces a rightmost derivation in reverse for a valid input string [1]

LR parsers belong to the class of "shift-reduce" parsers, so named because of how they operate They "shift" their input onto a stack and, at appropriate times, "reduce" the stack by recognizing the use of a particular grammar rule A reduction causes the stack symbols corresponding to the rule's right-hand side to be popped off the stack, and replaced by that rule's left-hand side

For example, consider the CFG in Figure 3 1 The behavior of a LR parser for this grammar on the input *aacbb* is shown in Figure 3 2 Looking at the reductions

$$
\begin{aligned}
S &\rightarrow a\ S\ B \\
S &\rightarrow c \\
B &\rightarrow b
\end{aligned}
$$

Figure 3.1 A simple CFG

Stack	Input	Action
$	aacbb$	shift a
$a	acbb$	shift a
$aa	cbb$	shift c
$aac	bb$	reduce by S → c
$aaS	bb$	shift b
$aaSb	b$	reduce by B → b
$aaSB	b$	reduce by S → a S B
$aS	b$	shift b
$aSb	$	reduce by B → b
$aSB	$	reduce by S → a S B
$S	$	accept

Figure 3.2 A LR parser trace

made by the parser, one can confirm that it has discovered the derivation

$$S \underset{rm}{\Rightarrow} aSB \underset{rm}{\Rightarrow} aSb \underset{rm}{\Rightarrow} aaSBb \underset{rm}{\Rightarrow} aaSbb \underset{rm}{\Rightarrow} aacbb$$

in reverse order

How does a LR parser decide what actions to take? LR parsers actually look for handles — a handle can be thought of as the right-hand side of a grammar rule,

Rule	Left Contexts	Viable Prefixes
S → a S B	ϵ, a, aa, aaa, ...	ϵ, a, aS, aSB, aa, aaS, aaSB, ...
S → c	ϵ, a, aa, aaa, ...	ϵ, c, a, ac, aa, aac, ...
B → b	aS, aaS, aaaS ...	ϵ, a, aS, aSb, aa, aaS, aaSb, ...

Table 3.1 Left contexts and viable prefixes

but *only* when reduction to the rule's left-hand side would correspond to a rightmost derivation step of the input [1]. In the grammar of Figure 3.1, c is a handle of the input ac, but b is not a handle of ab. Formally, if $A \rightarrow \alpha$ is a grammar rule and $S \overset{*}{\underset{rm}{\Longrightarrow}} \beta A w \underset{rm}{\Longrightarrow} \beta \alpha w$, then α is a handle at β. Under these circumstances, β is referred to as a left context of $A \rightarrow \alpha$, and any prefix of $\beta\alpha$ is called a viable prefix. Table 3.1 shows some left contexts and viable prefixes for the grammar of Figure 3.1.

Notice that the LR parser makes extensive use of its stack as temporary memory to recall what symbols it has seen and the order they occurred in. In terms of computational power, LR parsers may be modeled by push-down automata, finite automata augmented with a stack. Using the above terminology, a LR parser employs an automaton to find handles, and keeps track of viable prefixes on the stack.

Most current LR parsers are table-driven. The automaton's transitions and parser actions are encoded into tables, thus a short algorithm such as the one in Figure 3.3 is sufficient to drive the parser. Figure 3.2 notwithstanding, this algorithm reflects the fact that LR parsers actually maintain a stack of state numbers rather than grammar

symbols, this difference does not affect parser operation in any material way. This is because ' parse states encode the symbol that has been shifted and the handles that are currently being matched ' [10, page 139]

As with most types of parser, a LR parser can be made to accept a larger set of languages by allowing it to look ahead at symbols in the input [12], intuitively, this allows the parser to look into the future and choose parsing actions based on this foreknowledge. A LR parser using k symbols of lookahead is a LR(k) parser. Unless stated otherwise, only LR parsers and parse tables without lookahead will be considered in the remainder of this thesis — in other words, LR(0) parsers.

See [1] for a more thorough treatment of LR parsing and parsing in general.

```
function action(inputSymbol, state) {
    Based on its parameters, returns one of
        SHIFT n
        REDUCE A → α
        ACCEPT
        ERROR
    This would typically be a simple table lookup
}

function goto(nonterminalSymbol, state) {
    Based on its parameters, returns a state number to
    go to   Again, this is typically a table lookup
}

initialize stack to contain the start state

while (true) {
    input = lookAtNextInputSymbol()
    switch (action(input, topOfStack)) {
        case SHIFT n
            push n
            consumeInputSymbol()
        case REDUCE A → α
            pop |α| states from stack
            push goto(A, newTopOfStack)
        case ACCEPT
            accept input
        default
            error
    }
}
```

Figure 3.3 Table-driven LR parsing algorithm

Chapter 4

Reducing LR Parsing Overhead

To achieve our goal of building faster Tomita parsers, we begin by reducing the amount of overhead consumed by their inner workings — LR parsers.

4.1 Previous Work

Much attention has been devoted to speeding up LR parsers, and the majority of this research pertains to implementation techniques. The argument is that interpreted, table-driven programs are inherently slower than hardcoded, directly-executable programs, given that, the best way to speed up a table-driven LR parser is to convert it into a directly-executable form that needs no tables.

[32, 18, 34, 4] all start with a LR parser's handle-finding automaton and translate

it directly into source code — this source code can then be compiled[2] to create an executable LR parser. Basically, each state of the automaton is directly translated into source form using boilerplate code. This process tends to produce inefficient code, so these papers expend effort optimizing the source code output.

Several other papers [35, 36, 27, 28] have taken a slightly different approach, introducing a technique called recursive ascent parsing. Here, a LR parser is implemented with a set of mutually recursive functions, one for each state[3] in a table-driven LR parser's handle-finding automaton. To quote Grune and Jacobs [12, page 221],

> 'The key idea is to have the recursion stack mimic the LR parsing stack. To this end, there is a procedure for each state, when a token is to be shifted to the stack, the procedure corresponding to the resulting state is called instead.'

Unfortunately, all of the above work is of limited use when applied to a Tomita parser. LR parsers produce a *single* derivation for an input string. In terms of implementation, a LR parser only needs to keep track of a single set of information: the current parser state — what the parser is doing right now, and what it's done in the past. In a table-driven LR parser, this information is kept on an explicit stack. In a directly-executable LR parser, the information exists through a combination of the CPU's execution stack and program counter.

[2] Or assembled, as is the case in [32].

[3] Two functions per state are reputed to be required in [28].

In contrast, a Tomita parser produces *all* derivations for an input string. This means that a Tomita parser may need to keep track of multiple parser states concurrently. To construct a directly-executable Tomita parser, one would need to maintain multiple CPU stacks and program counters. Certainly this is possible, but the overhead in doing so and switching between them frequently would be prohibitive, at least on a uniprocessor architecture.

Once direct execution of Tomita parsers is ruled out, the obvious line of inquiry is to investigate speeding up table-driven LR (and thereby Tomita) parsers. Looking at the LR parsing algorithm and its operation, one source of improvement would be to reduce the reliance on the stack. Fewer stack operations would mean less overhead and should result in a faster parser.

If stack-related overhead is to be reduced, then the ideal situation is to have no stack at all. So instead of parsing with push-down automata, we would be using finite automata.

4.2 Parsing with Finite Automata

Theoretically, it is impossible to parse CFLs using finite automata. For example, consider the language $L = \{a^n b^n, n > 0\}$. Given a constant $k > 0$, one can easily

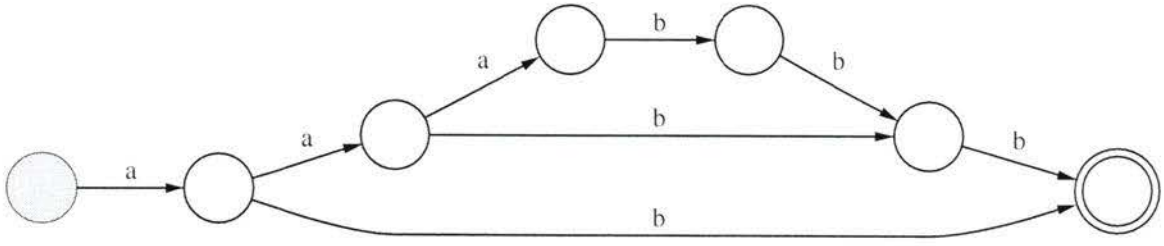


Figure 4 1 FA accepting $a^k b^k, 0 < k \leq 3$

construct a FA to recognize $a^k b^k$, such as the one in Figure 4 1 for $0 < k \leq 3$ Unfortunately, such a FA doesn't accept the input $a^{k+1} b^{k+1}$, despite the fact that that input is in L

In contrast, L *is* recognizable by a PDA each a is pushed onto the stack as it is read, the stack is popped once for each b read, and an empty stack must correspond to the end of input Effectively, the stack is used to count the number of a symbols seen

When parsing CFLs with a PDA, the stack is used to remember information In the previous example, it retained a single number, more generally, it can retain the entire left context of a handle Having no explicit means of storage, FAs cannot do this — their comparative lack of expressive power is colloquially stated as "finite automata can't count "

In practice, however, often a subset of a CFL is sufficient In the above example,

if we could determine that there was an upper bound u on n, then the language we are actually interested in is $a^n b^n, 0 < n \leq u$. This new language is recognizable with a FA having $2u + 1$ states.

The same principle holds true for programming languages. Compiler writers often set limits on a programming language for implementation reasons. Some examples limiting the complexity of arithmetic expressions, restricting the depth that blocks or functions can nest, limiting the number of labels in a `case` statement. The net effect of imposing such restrictions in a compiler is that the compiler no longer accepts the full language as specified by the language's grammar. That being the case, it is reasonable to consider eliminating a LR parser's stack, and instead constructing a large FA to parse the restricted language.

Similar ideas have been explored in the natural language community. [33] uses a FA to recognize an approximation of a CFG, by design, their FA accepts a superset of the original language. However, since we are trying to speed up LR parsing, it is important not to accept any inputs not accepted by the original parser. Given this, and the fact that language subsets arise naturally in practice, our work only considers using FAs to accept subsets of CFGs.

4.3 Recursion in Grammars

How do we construct our parsing FAs? A discussion of the exact method is deferred to Section 4.5. As it turns out, some grammar transformation is needed before the FA can be built. In this section and the next, we motivate the need for this transformation and describe it in detail.

The problem in constructing parsing FAs (PFAs) comes from recursion in the grammar. A grammar is recursive if, for any nonterminal A, $A \stackrel{*}{\Longrightarrow} \beta A \gamma$. Left recursion is the case where $A \stackrel{*}{\Longrightarrow} A\gamma$, right recursion is where $A \stackrel{*}{\Longrightarrow} \beta A$.

We will examine three cases: left recursion, right recursion, and "other" recursion (recursion which is neither exclusively left nor right). It is assumed, without loss of generality, that trivial recursion of the form $A \stackrel{*}{\Longrightarrow} A$ is not present in the grammar.

4.3.1 Left Recursion and PFAs

Left recursion is trivial to handle. In a LR parser, left recursion yields a shallow stack — a handle is accumulated atop the stack and is reduced away immediately. A similar process happens with PFAs. With a PFA, the handle of a left-recursive rule is recognized, and a reduction causes a simple state transition.

To illustrate, the grammar in Figure 4.2 contains the left-recursive rule S → S a. This grammar, which generates the language ba^*, is easy to represent with a PFA.

S → S a
S → b

Figure 4.2: A left-recursive grammar

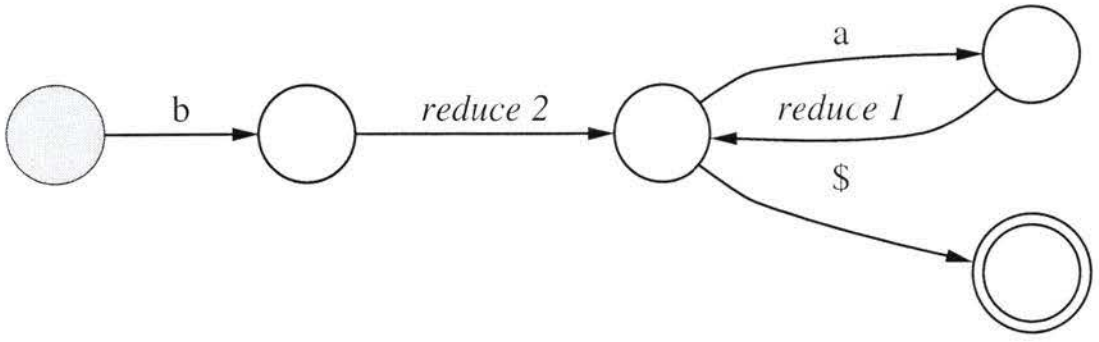


Figure 4.3: PFA for a left-recursive grammar

because it only needs to remember a small, finite amount of information: has a *b* been seen? was an *a* just seen? have we just seen the end of input? The reductions are straightforward too: If a *b* is seen, the PFA can immediately reduce by S → b, if an *a* is seen, the PFA can reduce by S → S a immediately. The PFA for this grammar is shown in Figure 4.3.

Compared to a FA, there are two unusual aspects to the PFA which warrant explanation:

- The end-of-input symbol, $, is explicitly represented in the PFA, even though

it was not in the grammar. This is because the PFA is based on an augmented grammar having a new start symbol S' and a new grammar rule $S' \rightarrow S\ \$$.

- Edges are labeled with reduction actions. Making a transition across one of these reduction edges does not cause any input to be consumed, but it does indicate that the PFA is performing a reduction by a grammar rule. Grammar rules are referred to by number, so *reduce 2* means a reduction by the second rule in the grammar. Figure 4.4 shows the augmented grammar complete with rule numbers — further grammars will be shown in this manner when appropriate.

Formally, a PFA inherits much from the definition of a FA [29]. A PFA is a five-tuple $M = (Q, \Sigma, \Delta, s, f)$, where

Q is a finite set of states,

Σ is the input alphabet,

$s \in Q$ is the start state,

$f \in Q$ is the accepting state,

and Δ is a transition relation, a finite set whose members are in $Q \times (\Sigma \cup R \cup \{\perp\}) \times Q$.

R is a finite set of symbols that represent reduction by grammar rules — there is one distinct symbol in R per rule, and $R \cap \Sigma = \emptyset$. (The symbols in R are the formal equivalent of *reduce n*.) The purpose of the symbol $\perp$ is explained in Section 4.4, for

$$\begin{array}{llll} 0 & S' & \rightarrow & S\ \$ \\ 1 & S & \rightarrow & S\ a \\ 2 & S & \rightarrow & b \end{array}$$

Figure 4.4 An augmented left-recursive grammar

now, it is enough to know that $\perp \notin (R \cup \Sigma)$. There is only a single accepting state as a result of augmenting the grammar: there is a unique way to make a transition on the end-of-input symbol.

4.3.2 Right Recursion

Now consider a right-recursive grammar, such as the one in Figure 4.5, and a valid input string such as *aaab*. The derivation of that input string is

$$S' \underset{rm}{\Rightarrow} S\$ \underset{rm}{\Rightarrow} aS\$ \underset{rm}{\Rightarrow} aaS\$ \underset{rm}{\Rightarrow} aaaS\$ \underset{rm}{\Rightarrow} aaab\$$$

which the PFA must produce in reverse — herein lies the problem.

Assume for the moment that we can construct a PFA for this grammar. Figure 4.6 shows the actions taken by this hypothetical PFA. (Empty action fields mean that nothing happens aside from a PFA state transition.) The sequence of consuming input and performing reductions must occur in this order for the PFA to find the

0	S′	→	S $
1	S	→	a S
2	S	→	b

Figure 4.5 A right-recursive grammar

correct derivation. Why? For this grammar, a PFA must see the entire input before announcing any reductions, and when it does, it must have one *reduce 1* for every *a* it read. In many respects, this is the same problem as recognizing $a^n b^n$ with a FA in the previous section — it can't be done. The key here is that the PFA must *remember* what it has seen.

In parsing terms, our hypothetical PFA is looking for the handle *b*, and the PFA must recognize and remember *b*'s entire left context in order to find the right reductions. Unfortunately, for the full language, the set of left contexts is infinite: $\{\epsilon, a, aa, aaa, \ldots\}$. Compare this to the left-recursive example, where the set of left contexts was always $\{\epsilon\}$.

4.3.3 Other Recursion

For completeness, grammars that contain other types of recursion must be considered. These types of recursion are sometimes unavoidable, as in the arithmetic expression grammar of Figure 4.7, the Dyck languages [26], or the `if-then-else` construct.

Input	Action
aaab$	
aab$	
ab$	
b$	
$	reduce 2
$	reduce 1
$	reduce 1
$	reduce 1
	accept

Figure 4.6 A PFA trace of *aaab*

Stmt → if Expr then Stmt else Stmt

In actual fact, the problem that other forms of grammar recursion pose is exactly the same one presented by right-recursive grammars. To produce the correct derivation, the PFA would have to remember an infinite set of left contexts.

For example, in the grammar of Figure 4.7, a PFA would need to keep count of the number of left parentheses seen. Otherwise, it would not be able to look for the correct number of matching right parentheses. Again, we are back to the $a^n b^n$ problem.

$$
\begin{array}{llll}
0 & S' & \rightarrow & E\ \$ \\
1 & E & \rightarrow & E + F \\
2 & E & \rightarrow & F \\
3 & F & \rightarrow & (\ E\ ) \\
4 & F & \rightarrow & n
\end{array}
$$

Figure 4.7: A grammar for simple arithmetic expressions.

4.4 Grammar Expansion

To solve the problems posed by non-left-recursive grammars, we apply the ideas from Section 4.2. What we want to do is to make finite the set of left contexts that the PFA must recognize and remember. In terms of the grammar, we want to limit its depth of recursion: for example, one can imagine a grammar for arithmetic expressions where parenthesized expressions may not be nested greater than ten deep.

A straightforward approach to limiting grammars [15] is to essentially "unroll" the recursion in the grammar. An example is shown in Figure 4.8. We use the notation $A!n$ to indicate that the nonterminal A should be expanded n times. The point in the grammar where expansion occurs is called a limit point.

When a grammar has been fully expanded, the nonterminal at the limit point is replaced by the special symbol $\perp$. Upon encountering $\perp$ in the transition relation, the PFA outputs an error message and rejects the input. A PFA for the expanded grammar of Figure 4.8 would accept inputs b, ab, and aab, but output an error for

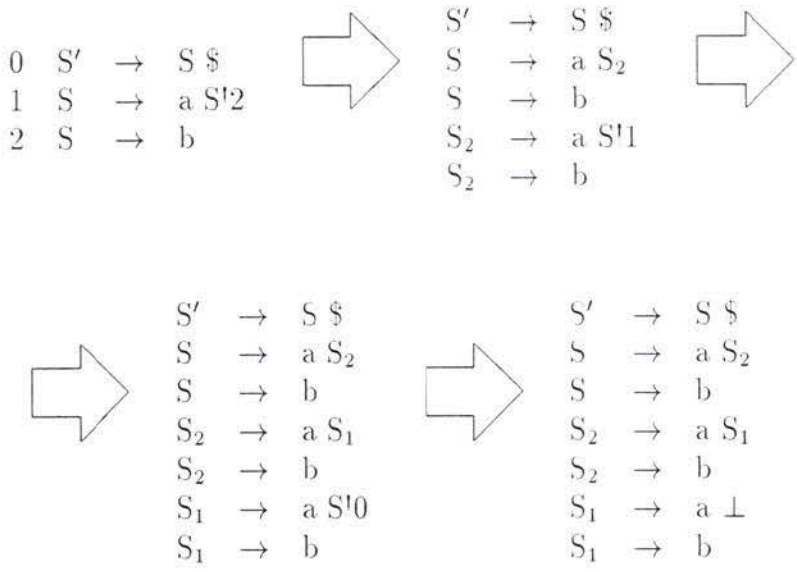


Figure 4.8 Grammar expansion

aaab

We devised the following algorithm to expand the grammar

1. Choose a rule $r_s = \mathrm{A} \rightarrow \alpha\ \mathrm{B}!n\ \beta$

2. If $n = 0$, replace r_s with $\mathrm{A} \rightarrow \alpha\ \perp\ \beta$

3. If $n > 0$

   (a) For each rule $r = \mathrm{C} \rightarrow \gamma$ such that $B \stackrel{*}{\Longrightarrow} \zeta C \eta$ (this includes the case where $B = C$), add a new rule r'. To map r into r', replace all nonterminal symbols D in r with D_n. If $r = r_s$, then the limit point $B!n$ should be

mapped into $B'(n-1)$ in r' instead

(b) Replace r_s with $A \rightarrow \alpha\ B_n\ \beta$

4 Repeat steps 1–3 until there are no more limit points to expand

For reduction purposes, a rule added through expansion should retain its parent's rule number. This is so reductions reported by a PFA make sense in terms of the original grammar.

4.5 Constructing the PFA

Once the grammar has been augmented and necessary grammar expansions have been performed, the PFA can be built.

4.5.1 Background

The theoretical basis for PFAs comes from some early work in LR parsing. In Knuth's seminal paper on LR parsing [24], he proposed two ways to determine if a grammar G was LR(k) for some integer $k \geq 0$.

1 Successfully construct a handle-finding automaton for G. This method directly yields a method for parsing the grammar (if the grammar is indeed LR(k)). LR parsers and research in LR parsing are almost exclusively based on this method.

2 Derive a new grammar F from G, then test the language generated by F for a specific condition. Only a few researchers [3, 13] have explored this method.

For our purposes, we are not concerned with whether or not G is LR(k), just with the construction method itself. Using the second method above, the following sections describe our technique for reducing LR parsing overhead.

4.5.2 The Derived Grammar

As mentioned, we need to derive a new grammar F from G [3]. Both terminals *and* nonterminals in G are treated as terminal symbols in F. F has a different set of nonterminals, which are derived from the nonterminals of G. The notation [A] is used to refer to a nonterminal in F which was derived from the nonterminal A in G.

Initially, F consists of the single rule

$$[S'] \rightarrow \epsilon$$

Then, one rule

$$[A] \rightarrow [B]\ \alpha$$

is added to F for every rule $B \rightarrow \alpha\ A\ \beta$ in G. Finally, we delete direct cycles from F (e.g. $[A] \rightarrow [A]$), which does not change L(F) [12], but makes F easier to handle from an implementation point of view.

$$
\begin{array}{lcl}
[\mathrm{S}'] & \rightarrow & \epsilon \\
[\mathrm{E}] & \rightarrow & [\mathrm{S}'] \\
[\mathrm{F}] & \rightarrow & [\mathrm{E}]\ \mathrm{E}\ + \\
[\mathrm{E}] & \rightarrow & [\mathrm{F}]\ (
\end{array}
$$

Figure 4.9 A derived grammar

Figure 4.9 shows F for the grammar in Figure 4.7 (page 29). The rule [E] $\rightarrow$ [E] has been deleted.

4.5.3 Properties of the Derived Grammar

F has one extremely useful property. Take $L_F([A])$ to mean L(F) where [A] is used as the start symbol. Then the set of left contexts for a rule A $\rightarrow \alpha$ in G is $L_F([A])$ [3]. We will refer to this set of left contexts as LC(A).

To understand why, recall the definition of a handle in Chapter 3. If

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \beta A w \underset{rm}{\Longrightarrow} \beta \alpha w$$

then α is a handle at β. Going back a step further, A must also be part of a handle. Say $\beta = \beta_1\beta_2$, $w = w_1w_2$, and B $\rightarrow \beta_2$ A w_1 is a rule in G. Then we have

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \beta_1 B w_2 \underset{rm}{\Longrightarrow} \beta_1\beta_2 A w_1 w_2 \underset{rm}{\Longrightarrow} \beta_1\beta_2\alpha w_1 w_2$$

This means that $LC(B) \subseteq LC(A)$. More specifically, because B $\rightarrow \beta_2$ A w_1, we can strengthen it to say $LC(B)\ \beta_2 \subseteq LC(A)$. ($LC(B)\ \beta_2$ is written as shorthand for $\{\gamma\beta_2 | \gamma \in LC(B)\}$.)

Using a grammar as a generative device, we can express the above relation between LC(B) and LC(A) with the rule [A] $\rightarrow$ [B] β_2 in F. The rule [S'] $\rightarrow \epsilon$ is added to F because the left context of G's start symbol must be ϵ.

Given that $L_F([A]) = LC(A)$, the handle of A $\rightarrow \alpha$ can be found by looking for $L_F([A])\ \alpha$ from the start of the input string, call this set $LRC(\text{A} \rightarrow \alpha)$. This ability to find handles gives us a means to parse input. But can it be done with a FA?

Another property of F is that it is a regular grammar by construction[4] [3]. As a result, $L_F([A])$ is regular for all nonterminals $[A]$. The right-hand sides of G's grammar rules, as strings, form a regular language as well. So the set $\Phi = \left\{\bigcup_{\text{A} \rightarrow \alpha \in G} LRC(\text{A} \rightarrow \alpha)\right\}$ of all G's handles and their left contexts is regular due to closure properties of regular languages [14]. It is therefore sufficient to use a FA to find handles and their left contexts — not a surprising result!

As we realized, what *is* different is that F gives us a systematic method for generating all left contexts of a handle. This way, all handles and all their left contexts can be enumerated in the PFA (due to grammar expansion, there will be a

[4]To be precise, F is leftlinear [14, 3]

finite number of them) If we did not have a separate path in the PFA for each, then the PFA would have no idea what handles to start looking for following a reduction

4.5.4 PFA Construction Algorithm

Now that we have a theoretical basis for the PFA, we can describe our algorithm for its construction Starting with an augmented, expanded CFG G, create the derived grammar F, and the set Φ Then, the algorithm is

1 Choose a unique start state s for the PFA For all members $X_1X_2 \ldots X_n \in \Phi$, add to the PFA the *minimal* number of transitions needed so that there is a path $X_1X_2 \ldots X_n$ in the PFA starting at s No transitions on ϵ are added to the PFA

For the expanded grammar in Figure 4 8, $\Phi = \{\epsilon S\$, \epsilon aS_2, \epsilon b, aaS_1, ab, aaa\perp, aab\}$, and its PFA would look like Figure 4 10 after this step The result of this step is a trie data structure [25]

2 Now the reduction transitions can be added to the PFA Take all members $X_1X_2 \ldots X_mX_{m+1} \ldots X_n \in \Phi$, where $X_{m+1} \ldots X_n$ is a handle of the rule $A \rightarrow X_{m+1} \ldots X_n$ at point $X_1X_2 \ldots X_m$

Let q_0 be the state at the end of the path $X_1X_2 \ldots X_mX_{m+1} \ldots X_n$ starting with

s, q_1 is the end state of the path $X_1 X_2 \ldots X_m A$, also starting with s. Assuming the rule $A \rightarrow X_{m+1} \ldots X_n$ is numbered k, then add a transition from q_0 to q_1 labeled *reduce* k. As a special case, the final state f for the PFA is the state at the end of the path $S\$$.

Figure 4.11 shows the PFA from Figure 4.10 with its reduction edges added.

3. Delete transitions in the PFA that are labeled with a nonterminal symbol from G. They are superfluous, since the PFA can never read nonterminals from an input string. The final PFA for our running example is shown in Figure 4.12.

4.6 Choosing Limit Points Automatically

After implementing our algorithm from the previous section, we tried to construct PFAs for increasingly larger grammars. As we did so, it became increasingly difficult to select appropriate limit points by inspection. Some way to have limit points suggested automatically was needed.

Since the goal of limit points is to ensure that Φ is finite, the first step was to determine what could make Φ's size infinite to begin with. This is quite easy: there are a finite number of grammar rules in G, so the set of handles must be finite. Therefore, if Φ is infinite in size, it must be because of the set of left contexts.

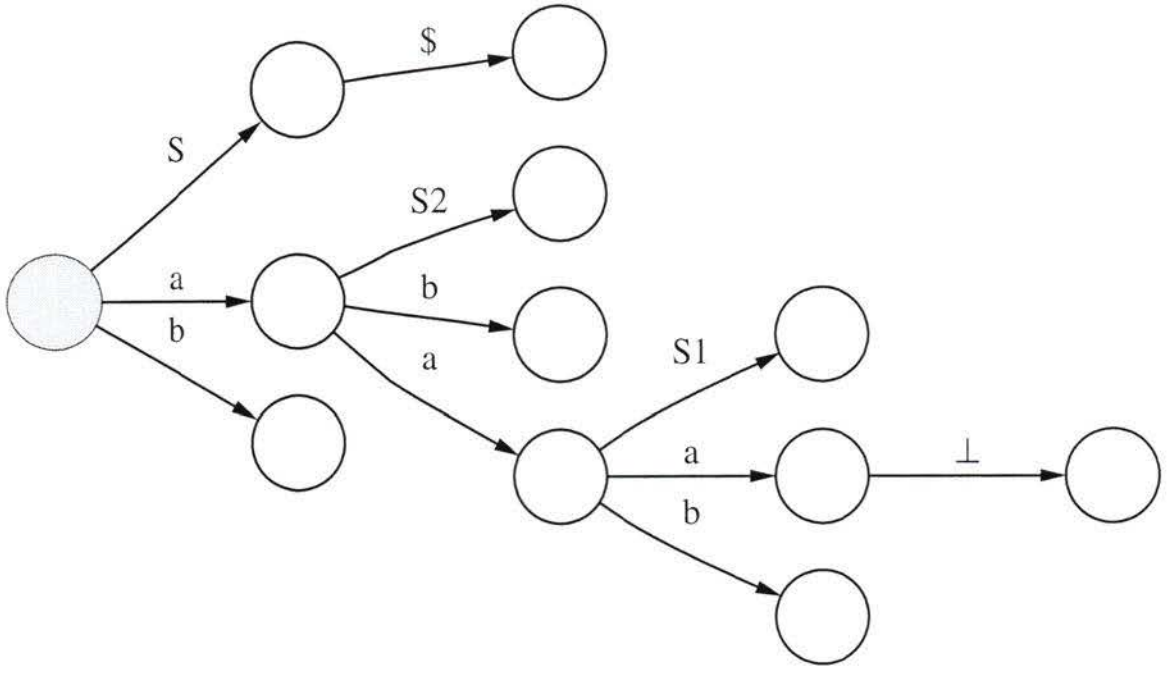


Figure 4.10: PFA after Step 1

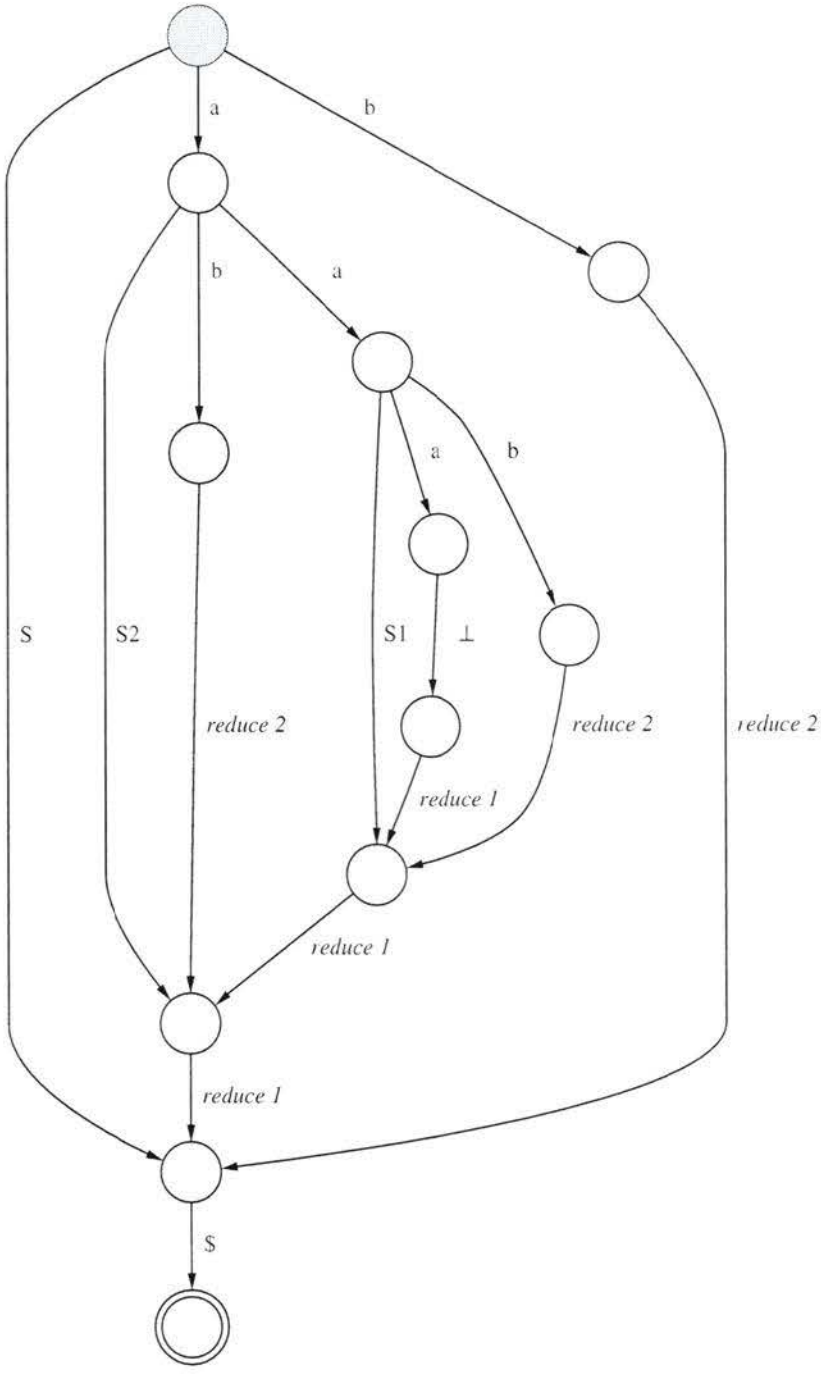


Figure 4.11 PFA after Step 2

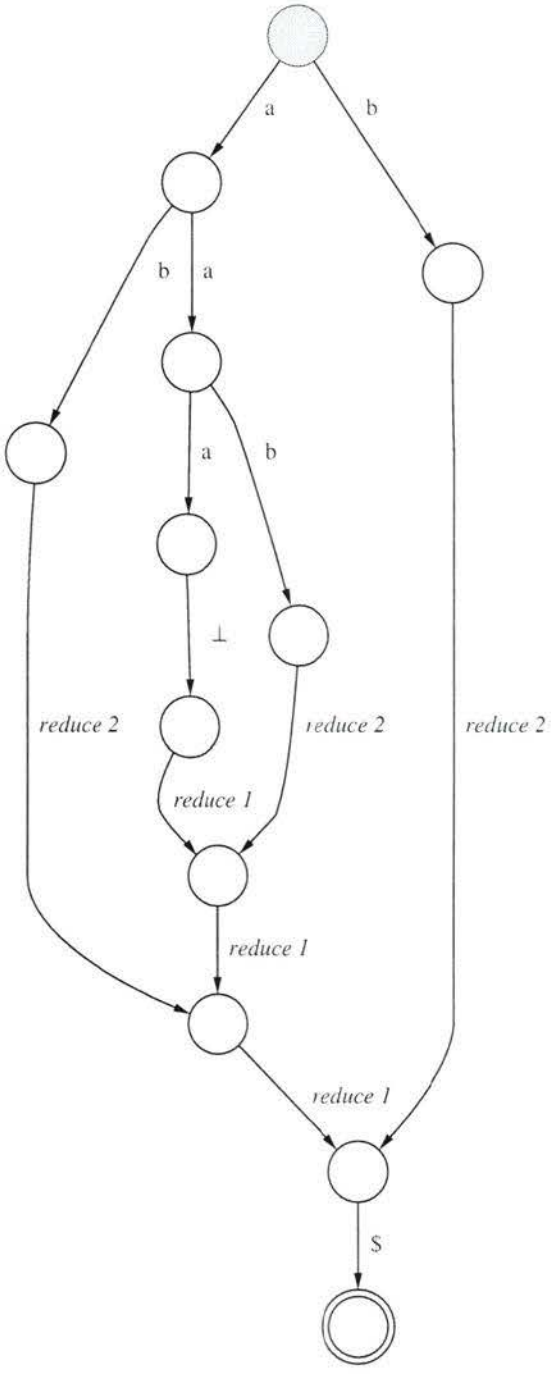


Figure 4.12 PFA after Step 3

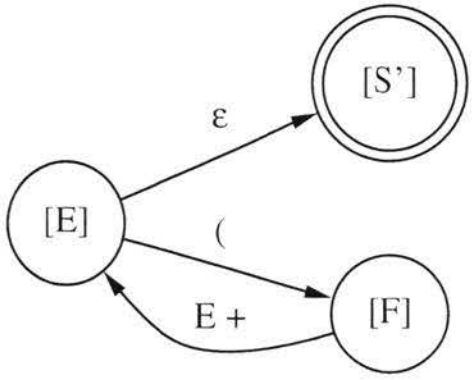


Figure 4 13 Derived grammar's equivalent FA

Since the left contexts are generated using the derived grammar F, analysis of it should yield a set of limit points Ideally, we would also like a minimal set of limit points — this would allow the PFA to accept the largest subset of the original (unexpanded) grammar's language

Because F is a regular grammar, it is equivalent to a FA [14] — call it F_{FA} The FA for the derived grammar of Figure 4 9 is shown in Figure 4 13

We also know that a FA's transition diagram contains a (non-ϵ) cycle if and only if the FA accepts an infinite language [14] So if we remove a set of transitions from F_{FA} such that it no longer contains any cycles, then the language accepted/generated by it must be finite This set would provide us with the limit points

The problem of removing a minimal set of edges from a directed graph so that

Ada	42
ANSI C	38
Java	23
Modula-2	23

Table 4.1 Limit points derived heuristically

the resulting graph has no cycles is well-known in graph theory the feedback arc set (FAS) problem [37] Unfortunately, the FAS decision problem is NP-complete [22], and the corresponding optimization problem — finding the minimal FAS — is NP-hard [11] There are, however, heuristic algorithms for the problem We have chosen to implement the algorithm from [8] due to its relative simplicity

The number of limit points obtained for various programming language grammars is shown in Table 4.1 It is important to remember that these numbers may be lower, depending on F_{FA} and the heuristic algorithm For example, starting with the computed limit points, hand experimentation revealed that no more than twelve limit points are needed for the Modula-2 grammar

Clearly, the work in this thesis will directly benefit from further work on the FAS problem

4.7 Incorporating a Stack

One obvious drawback to the PFA so far is that it only recognizes a subset of the original unexpanded grammar. To remove this restriction, a stack is added to the PFA to form a parsing pushdown automaton (PPA).

How can a stack be incorporated into a PFA? Intuitively, the places where a grammar G is expanded are the natural places to push information onto a stack [16]; when a $\perp$ transition appears, essentially the PFA is stating that it no longer has a sufficient number of states to remember any more. By pushing information at those points, a PPA is able to remember that which the PFA cannot.

A PPA is a simple extension of a PFA. To construct a PPA for G, we first build a PFA for G, PFA_G, in the usual manner. Then, while there are $\perp$ transitions in PFA_G, choose one and do the following:

1. Find the nonterminal $S_\perp$ that was initially expanded and which caused $\perp$ to be placed in PFA_G. In Figure 4.12, $\perp$ appears as a result of the initial expansion of S (see Figure 4.8).

2. Create a new grammar $G_\perp$ from G. Initially, all rules in G are placed in $G_\perp$. Then, set the start symbol for $G_\perp$ to be $S_\perp$, and remove all rules from $G_\perp$ that are not reachable from this new start symbol. Augment $G_\perp$ with the rule

$S'_\perp \rightarrow S_\perp$ *pop* Due to the simplicity of the grammar in Figure 4.8, the net effect of this step is to replace $S' \rightarrow S\ \$$ in $G_\perp$ with $S'_\perp \rightarrow S_\perp$ *pop*

3. Construct a PFA for $G_\perp$, call it $PFA_\perp$. The states of PFA_G and $PFA_\perp$ must be disjoint. $PFA_\perp$ will act as a "subroutine" for PFA_G in the sense that when PFA_G reaches the $\perp$ transition, it will push a "return state" onto a stack, then go to $PFA_\perp$'s start state. When $PFA_\perp$ reaches a *pop* transition (which must be unique due to $G_\perp$'s augmentation), it goes to a state which is popped off the stack.

4. Say that the transition on $\perp$ in PFA_G was made from state q_0 to state q_1. Delete that transition from PFA_G, replace it with a transition from q_0 to the start state of $PFA_\perp$, and label the new transition *push* q_1.

5. Merge $PFA_\perp$ into PFA_G. Since these PPA construction steps continue while there are $\perp$ symbols in PFA_G, this means that all $\perp$ symbols in $PFA_\perp$ eventually get replaced.

The result of the above steps is the PPA for G. As all the $PFA_\perp$ "subroutines" are built independently of any left context seen by their "caller," they can be re-used in other contexts. So the maximum number of $PFA_\perp$ that will be created for G is bounded by the number of limit points. Also, notice that the construction of $G_\perp$

preserves any grammar expansion that has occurred in G

The final PPA for the grammar in Figure 4.8 is shown in Figures 4.14–4.15, complete with state numbers. A trace of the PPA on input *aaaaab* is shown in Figure 4.16.

The formal definition of a PPA resembles that of a pushdown automaton [29]. A PPA is a six-tuple $M = (Q, \Sigma, \Gamma, \Delta, s, f)$, where

Q is a finite set of states,

Σ is the input alphabet,

Γ is the stack alphabet ($\Gamma = Q$),

$s \in Q$ is the start state,

$f \in Q$ is the accepting state,

and Δ is a transition relation, a finite set whose members are in $Q \times (\Sigma \cup R \cup P) \times Q$. R has the same definition as it did for a PFA. P is a finite set, disjoint from Σ and R, whose members are in $(Q \cup \emptyset) \times (Q \cup \emptyset)$. P models stack operations: for example, $(\epsilon, 12)$ is a push and $(34, \epsilon)$ is a pop.

Table 4.2 shows some PPA sizes and the relatively small number of stack operations in each. The arithmetic expression grammar used is from Pfahler [34]; his automaton for the same grammar had twelve states. The Modula-2 grammar we used is recognizable using 386 states, according to the LALR(1) parser generator `yacc` [20]. In both grammars, the expansion factors at the limit points were all set to zero. As a

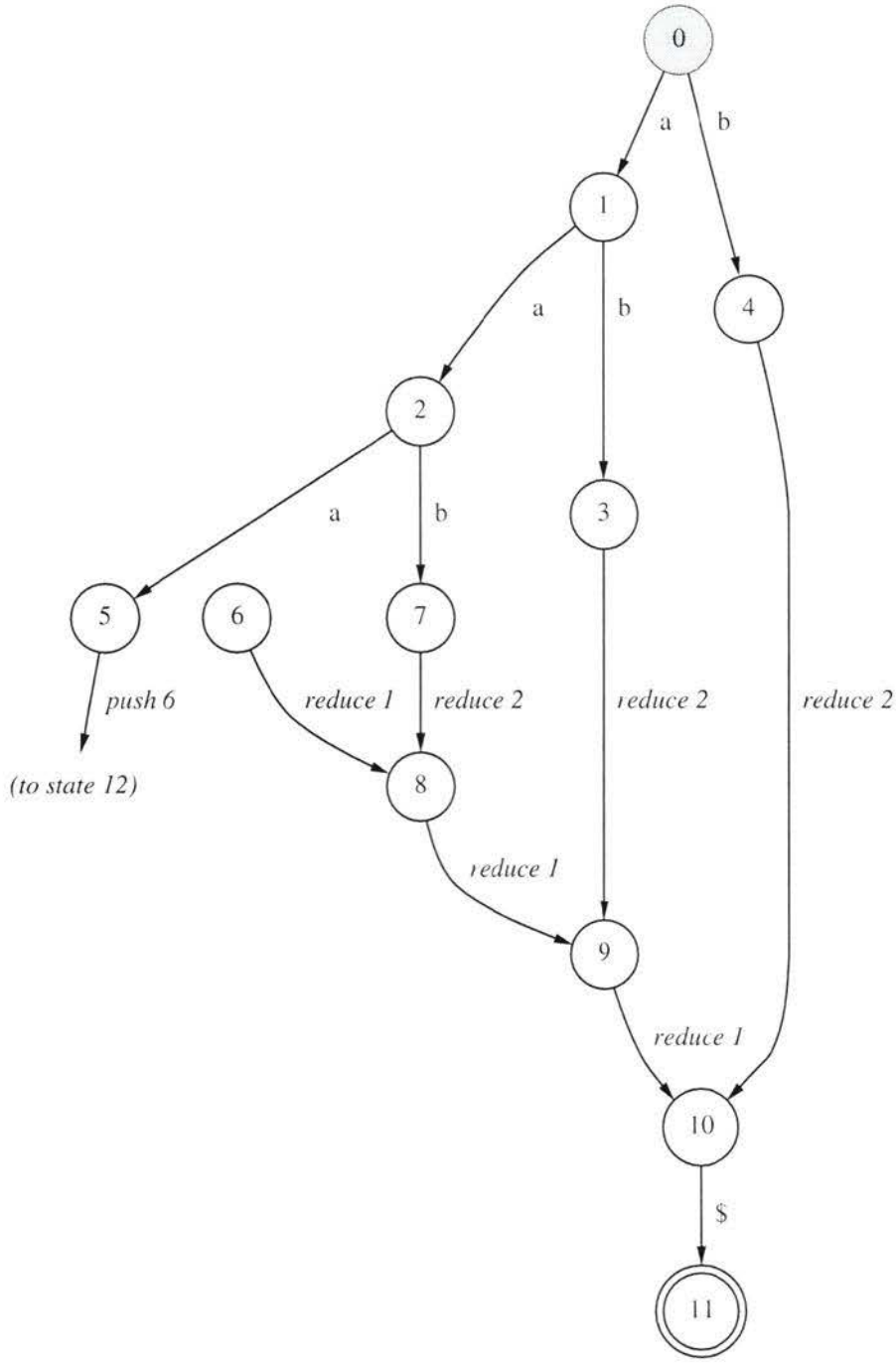


Figure 4.14 An example PPA

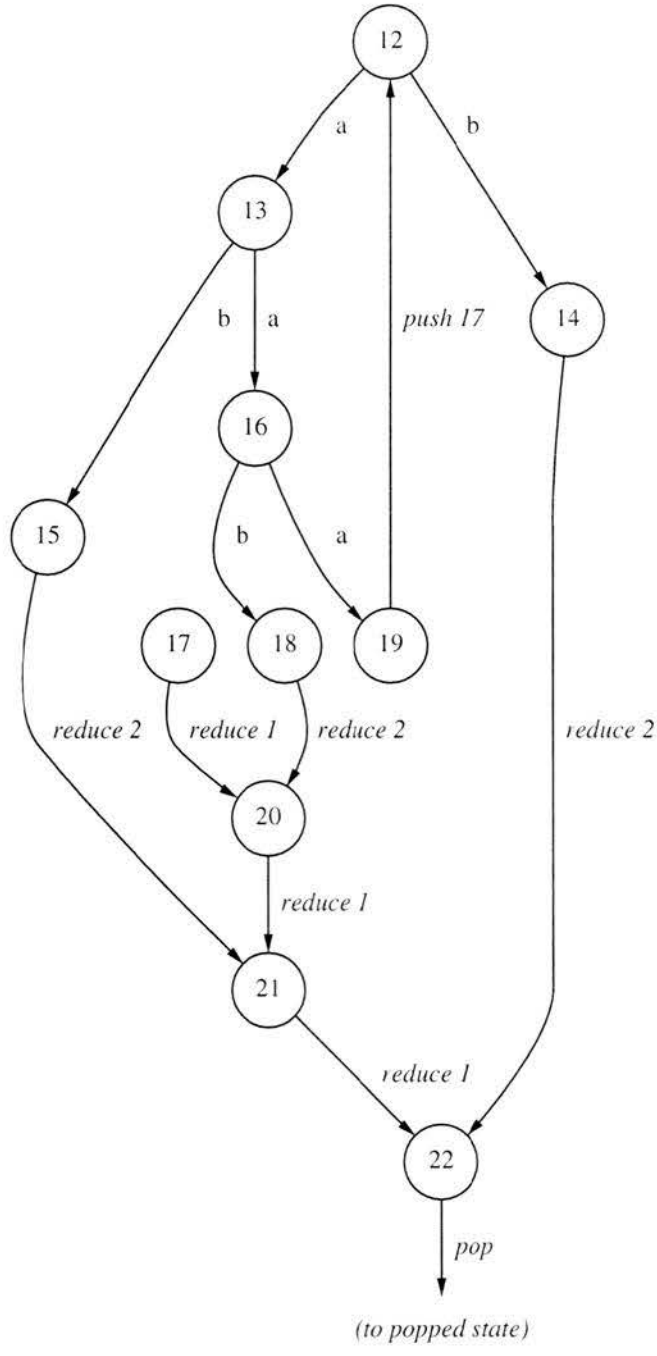


Figure 4.15 An example PPA (continued)

Stack	State	Input	Action
$	0	aaaaab$	
$	1	aaaab$	
$	2	aaab$	
$	5	aab$	push 6
$6	12	aab$	
$6	13	ab$	
$6	16	b$	
$6	18	$	reduce 2
$6	20	$	reduce 1
$6	21	$	reduce 1
$6	22	$	pop
$	6	$	reduce 1
$	8	$	reduce 1
$	9	$	reduce 1
$	10	$	
$	11		accept

Figure 4.16 PPA trace of *aaaaab*

	Pushes	Pops	States	Total Stack Operations
Modula-2	1928	8	18279	10.6%
Expression	8	1	56	16.1%

Table 4.2 Number of PPA states and stack operations

PPA must recognize more than a handle-finding automaton, it has many more states. While the PPAs may seem large, remember that we are trading space for time. With the proliferation of large, inexpensive memory in modern computers, the PPA size should not typically be a concern.

4.8 Modified LR Parsing Algorithm

The LR parsing algorithm, modified to use a PPA, is shown in Figure 4.17. (For comparison, the original table-driven LR parsing algorithm appeared in Figure 3.3.)

```
function action(inputSymbol, state) {
    Based on its parameters, returns one of
        SHIFT n
        REDUCE A → α, GOTO n
        PUSH m, GOTO n
        POP
        ACCEPT
        ERROR
    This can be implemented as a simple table lookup
}

initialize stack to be empty
currentState = start state

while (true) {
    input = lookAtNextInputSymbol()
    switch (action(input, currentState)) {
        case SHIFT n
            currentState = n
            consumeInputSymbol()
        case REDUCE A → α, GOTO n
            currentState = n
        case PUSH m, GOTO n
            push m
            currentState = n
        case POP
            currentState = state popped off stack
        case ACCEPT
            accept input
        default
            error
    }
}
```

Figure 4.17 Table-driven LR parsing algorithm using a PPA

Chapter 5

Application to Tomita's Algorithm

5.1 Background

Tomita's parsing algorithm, also known as generalized LR (GLR) parsing, was developed to parse natural languages efficiently [38] Tomita observed that grammars for natural languages were mostly LR, with occasional ambiguities

With that in mind, Tomita's algorithm behaves as a normal LR parser until it reaches a LR parser state where there is a conflict — the LR parser has a set of conflicting actions it could perform, and is unable to choose between them A Tomita parser is not able to choose the correct action either, and instead simulates nondeterminism by doing a breadth-first search over all the possibilities [12] Conceptually,

one can think of the Tomita parser reaching a conflict, and starting up a new parser running in parallel for every possible action, each new parser "process" would have a copy of the original stack. A parser process that finds what seems to be erroneous input may assume that the action it took from the conflict point was the wrong one and can terminate.

This cycle of a parser process starting others yields a wholly impractical algorithm. The time spent making copies of parser stacks could be enormous, not to mention the potentially exponential growth of the number of processes [40]. To address this, Tomita made two important optimizations:

1. A new process need not have a copy of its parent's stack. N processes can *share* a common prefix of a stack. From an implementation perspective, elements of the stack can all contain pointers to point to the previous element of the stack. Then, multiple stack elements can point to a common prefix. Figure 5.1 illustrates what happens to the stack when a conflict between three actions arises in state 56, the top of the stack is on the left side.

2. There are a finite number of automaton states the parser can be in. Several processes may be in the same state, albeit they may have different stack contents. A set of processes that are in the same state can merge their stacks together, leaving one resulting process. This places an upper bound on the number of

parsing processes that can exist

In a LR parser, its current state is the topmost state on the stack (see Figure 3.3 on page 17). So to merge N stacks, one would remove the top node from each — they must all have the same state number s — and create one node with state s that points to the remainder of the N stacks. In Figure 5.2, two stacks are merged together since their processes are both in state 17.

The result of these optimizations is called a graph-structured stack. (A slight misnomer, since the stacks actually form a directed acyclic graph.) The graph-structured stack in Figure 5.3, for instance, corresponds to four processes and five conceptual stacks.

To now understand how we have applied our PPAs to Tomita's work, we first examine what constitutes a state conflict in a PPA.

5.2 Characterization of PPA State Conflicts

Normal LR(0) parsers are subject to two types of conflicts [10]:

1. Shift/reduce conflicts, where both a shift action and a reduce action are possible from a single parser state.

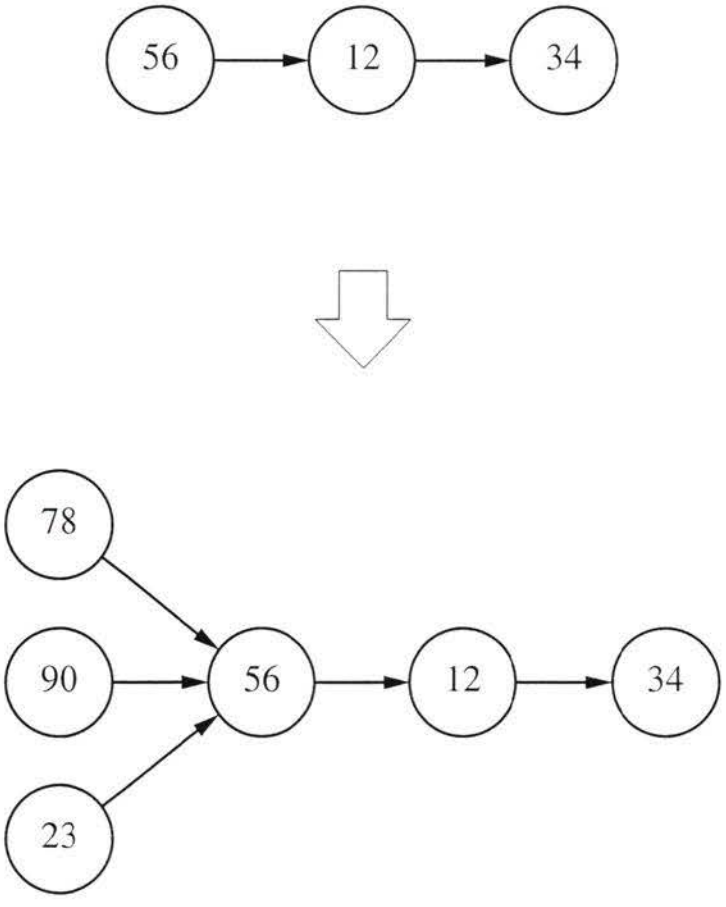


Figure 5.1 Stacks sharing a common prefix

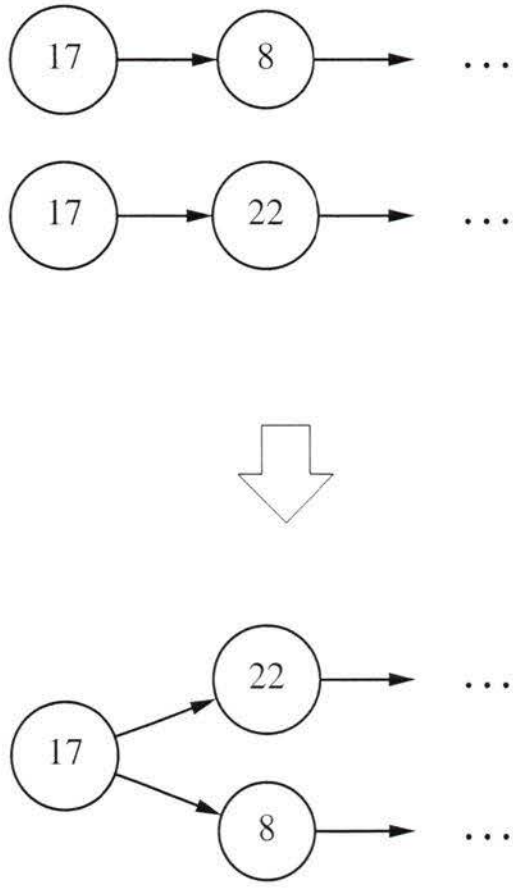


Figure 5.2 Stacks merging

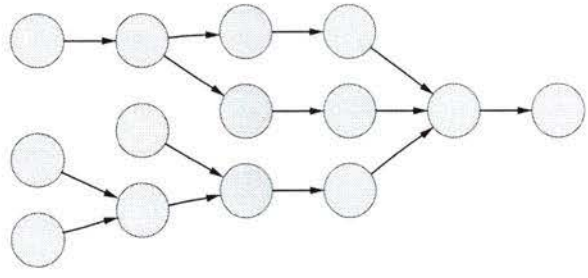


Figure 5.3 A graph-structured stack

	shift	reduce	push	pop
shift		X	X	X
reduce	X	X	X	X
push	X	X	X	X
pop	X	X	X	

Table 5.1 PPA state conflicts

2 Reduce/reduce conflicts, where two or more distinct reduction actions are possible from a single parser state

The other combination, shift/shift, cannot exist as a conflict because there can be only one shift edge for a given input symbol from a single state.

Recall that our PPAs are also LR(0) parsers, but we have more types of actions possible and hence more types of conflict that can arise. In fact, if a PPA state has any combination of two or more edges leaving it, there is a conflict unless all the edges are all "shift" actions labeled with terminal symbols, all *pop* actions. (Actually, multiple *pop* edges from a PPA state cannot occur if the PPA is built using the algorithm in Section 4.7.) Table 5.1 shows the various combinations, a 'X" in the table entry indicates a conflict.

Since a Tomita parser starts a new parsing process whenever it reaches an LR(0) conflict, a PPA-based Tomita parser also does so whenever it discovers one of the above PPA conflicts. While this may seem like a great deal more work, remember

that stack actions in a PPA are infrequent, in practice, most PPAs are unlikely to be riddled with conflicts.

5.3 PPA-Based Tomita Parsing

To use a PPA as the engine for a Tomita parser, we have devised an algorithm which is the combination of our modified LR parsing algorithm in Section 4.8 and the work of Tomita [38, 39, 40]. Its pseudocode is shown in Figures 5.5–5.6.

Some discussion of the data structures we used is appropriate. There are two major types of structures: one for processes, the other for stack nodes.

1. Processes. Each process structure has a PPA state number and a pointer to a stack top associated with it. Unlike LR parsers, a PPA's current state number is stored separately from the stack, so each process must maintain a PPA state number.

   Process structures are linked into one of two lists. The current process list contains the processes that still require processing for the current input symbol, the pending process list contains processes that will need processing when the next input symbol is read. Every time a new input symbol is read, the pending process list becomes the current process list.

2 Stack nodes. There are two types of stack nodes.

(a) Data nodes. This type of node contains the actual data of a process' stack. Each data node holds a single PPA state number, and a pointer to a previous stack node (i.e. pointing away from the stack top). If we used *only* this type of stack node, then we would have a tree-structured stack.

(b) Fan-in nodes. These nodes are used to make the graph-structured stack, each one contains a set of pointers to previous stack nodes. When two process' stacks are merged, a fan-in node is created which holds pointers to both stacks. In our implementation, to bound the amount of effort required to find a data node, we add the constraint that a fan-in node may only point to data nodes.

Figure 5.4 illustrates a sample parser configuration (the pending process list is not shown). Having now described the mechanics of our algorithm, the natural question is: how does it perform?

5.4 Empirical Results

We performed some timing experiments to compare a standard Tomita parser with our PPA-based Tomita parser. In the remainder of this section we discuss our experiments

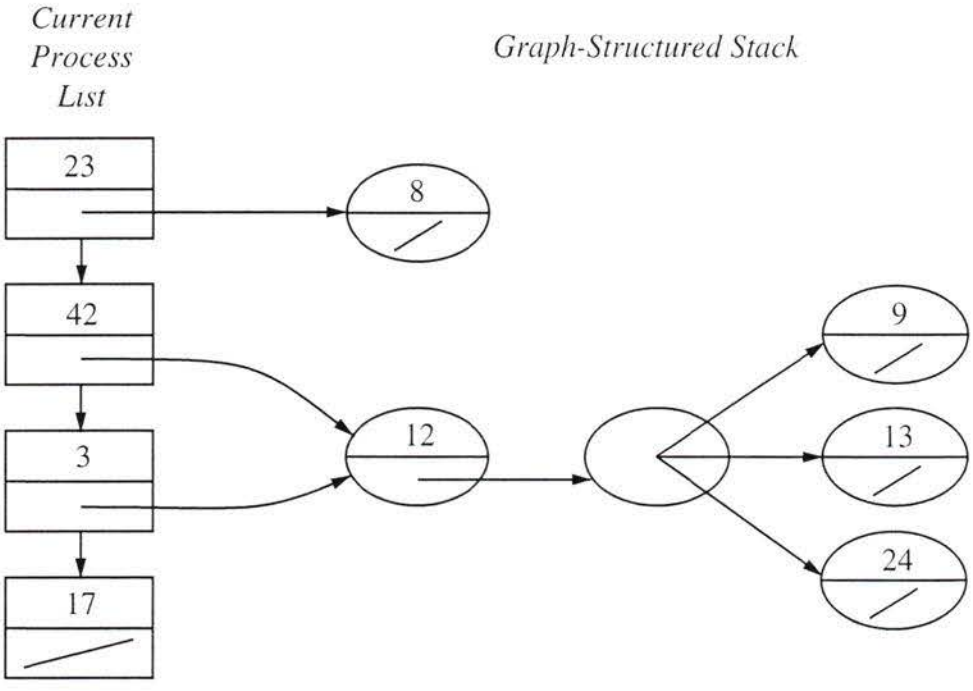


Figure 5.4 A sample parser configuration

```
function process(P, input) {
    foreach a ∈ action(input, P state) {
        switch (a) {
        case SHIFT n
            mergeIntoPending(n, P stack)
        case REDUCE A → α, GOTO n
            mergeIntoCurrent(n, P stack)
        case PUSH m, GOTO n
            mergeIntoCurrent(n, push(m, P stack))
        case POP
            let S be the set of stack data nodes atop P stack
            foreach node (state, stack) ∈ S {
                mergeIntoCurrent(state, stack)
            }
        }
    }
}

initialize pending process list to be empty
initialize current process list to be a single process,
    at the PPA's start state with an empty stack

while (current process list is nonempty) {
    input = getNextInputSymbol()
    while (current process list is nonempty) {
        remove a process P from the list
        process(P, input)
    }
    exchange the current and pending process lists
    if (input == EOF) {
        if (process in current process list is in accept state)
            accept input
        else
            reject input
    }
}
reject input
```

Figure 5.5 PPA-based Tomita parsing algorithm

```
function mergeIntoPending(state, stack) {
    Looks in the pending process list for a process with
    a matching state as that passed in.  If it finds such
    a process, it simply merges its stack with the one
    passed in, if not, it creates a new process structure
    with the given state number and stack pointer, and adds
    it to the pending process list
}

function mergeIntoCurrent(state, stack) {
    The same as mergeIntoPending(), but using the current
    process list instead
}

function push(state, stack) {
    Returns a new stack data node containing the given
    state and stack pointer
}

function action(inputSymbol, state) {
    Based on its parameters, returns a set containing
    zero or more of
        SHIFT n
        REDUCE A → α, GOTO n
        PUSH m, GOTO n
        POP
    This can be implemented as a simple table lookup
}
```

Figure 5.6 PPA-based Tomita parsing algorithm (continued)

and the results

5.4.1 Timing Particulars

As a basis for comparison, we used the public domain Tomita parser available from the `comp compilers` Usenet newsgroup archive[5]. It uses LR(0) parse tables internally which are computed at startup. Both it and our PPA-based Tomita parser are implemented in C.

To ensure that the timings reflect only parsing speed, we have extricated the lexical analyzer *and* the LR(0) parse table computation code from the public domain parser and used it in our PPA-based parser. In other words, our parser incurs the same startup penalty and lexical analysis overhead as the public domain parser. The only change we have made to the public domain parser's source code is to increase the size of a string table used by the lexical analyzer, this is so the lexical analyzer (as used in both parsers) would be able to handle our tests involving long input strings.

All tests were run on a Sun SPARCsystem 300 with 32M of RAM. Both parsers were compiled using `gcc` with compiler optimization (`-O`) enabled. To try and mitigate the effect of unpredictable system conditions on our timings, we ran the tests five times on each input, the results we report are the arithmetic mean of those times.

[5] `http //www iecc com` as of this writing

S → S a
S → b

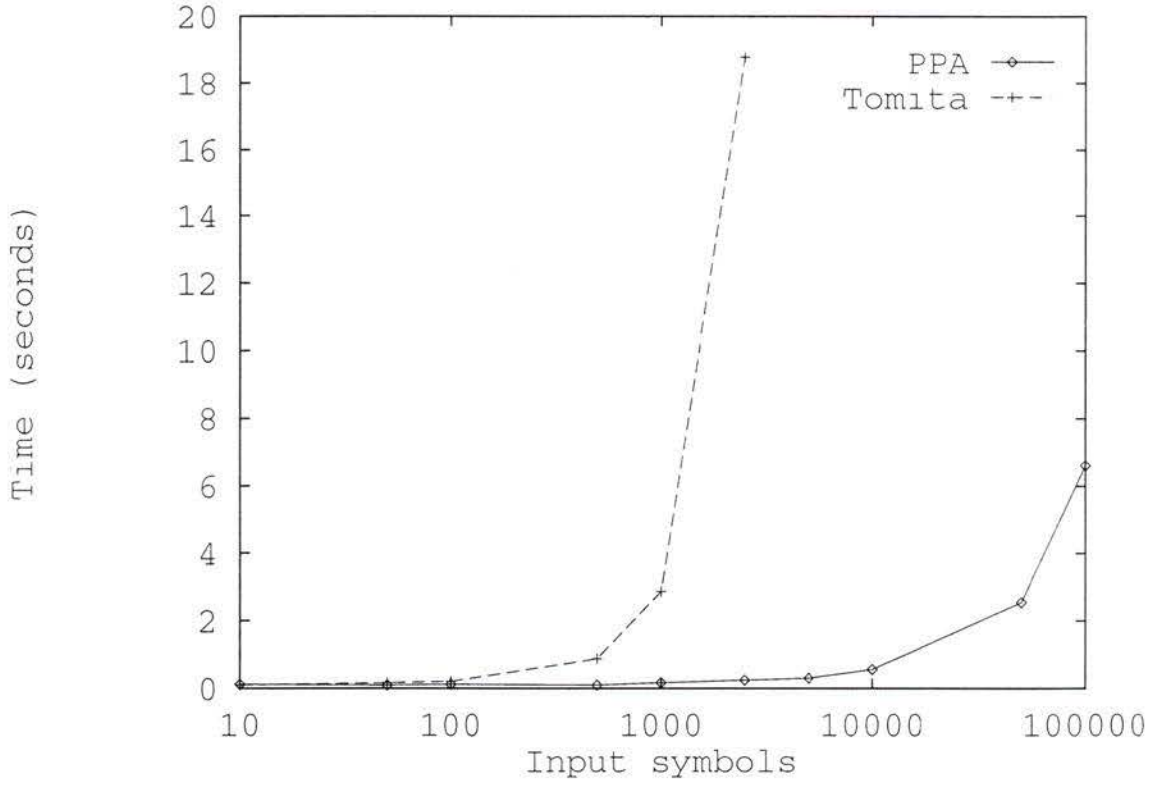


Figure 5 7 Timings for Grammar 1

Our results are shown in Figures 5 7–5 11 along with the grammars used For convenience of reference, we have numbered the grammars 1 through 5 The grammars have the limit points shown that were used for the PPA, they were of course not needed for the public domain parser

S → a S'0 b
S → c

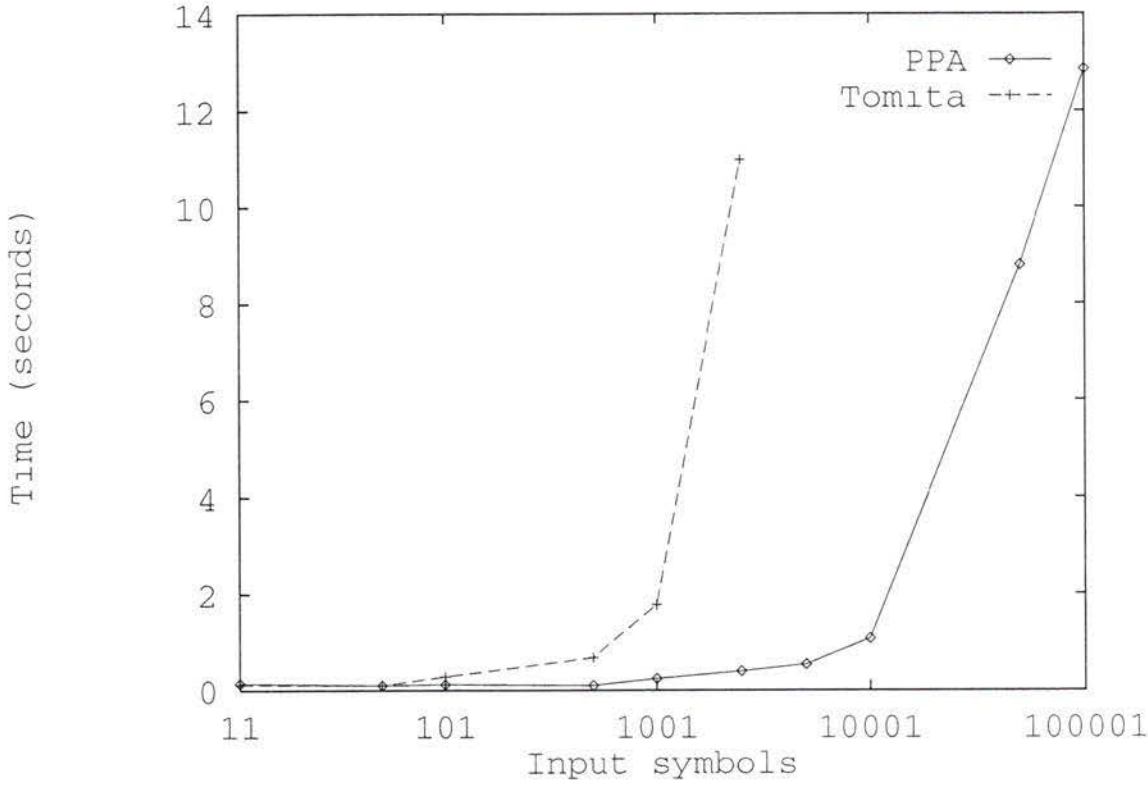


Figure 5.8 Timings for Grammar 2

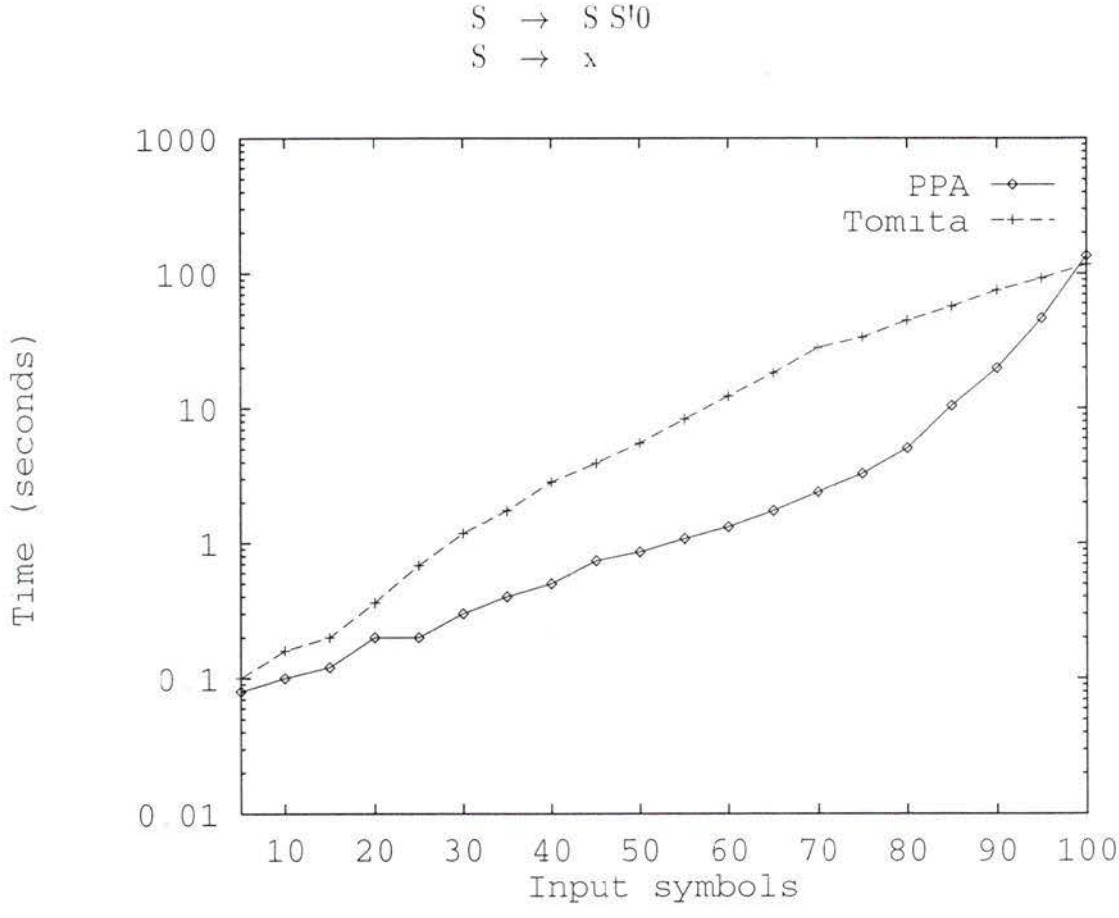


Figure 5.9 Timings for Grammar 3

S → S S'0 S'0
S → ε S'0
S → ε

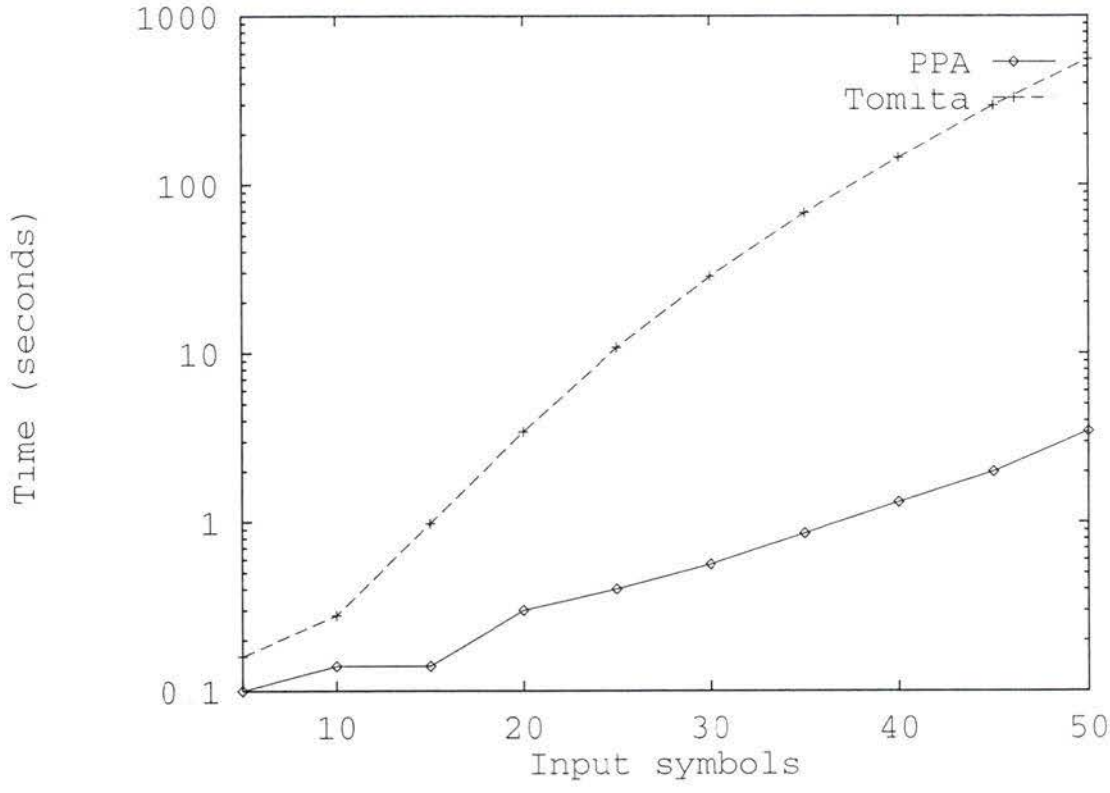


Figure 5.10 Timings for Grammar 4

$$S \rightarrow S\ S\ 'n$$
$$S \rightarrow \backslash$$

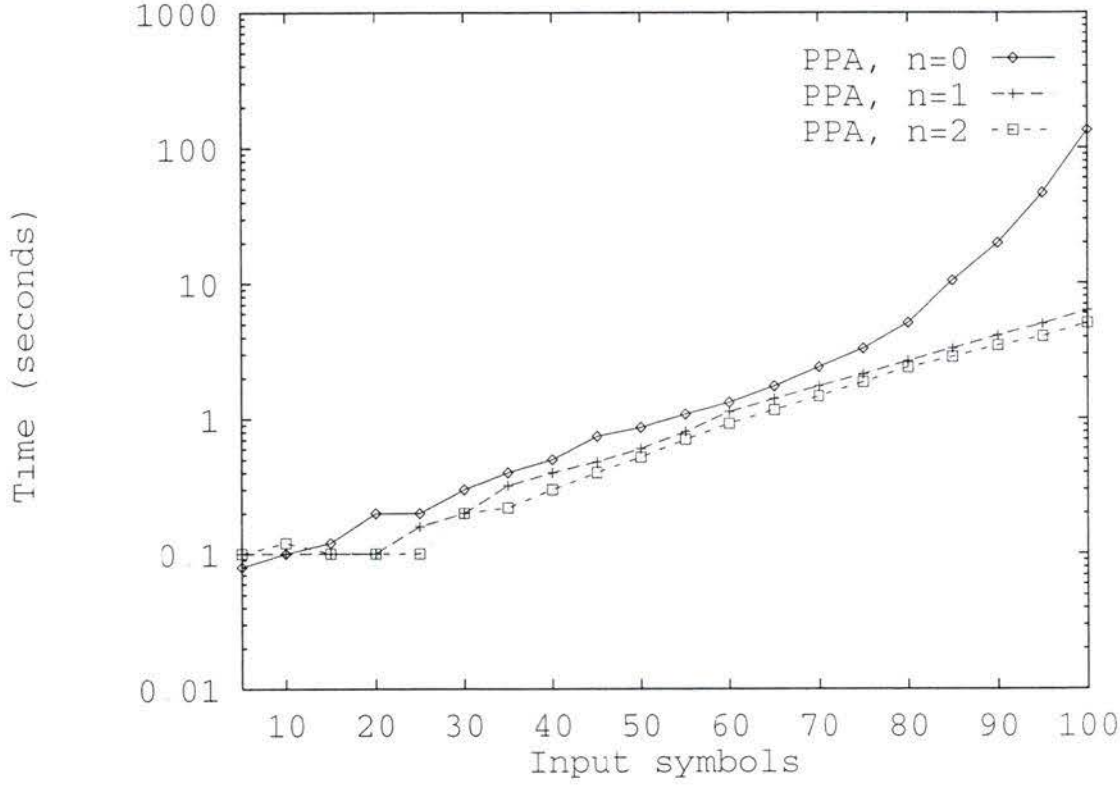


Figure 5.11 Timings for Grammar 5

5.4.2 Discussion of the Results

Grammars 1 and 2 are unambiguous, and were selected to illustrate several points about PPA operation. First, since these two grammars are unambiguous, any Tomita parser should exhibit run times that are linear with the input size, as our results show. (The public domain Tomita parser ran out of memory on samples larger than about 2500 symbols for both grammars after about thirty seconds, so only partial results are shown for it.)

Second, Grammar 1 is one of the best possible cases for a PPA. Because it is left recursive, the set of left contexts is ϵ, the PPA requires no stack operations and essentially spends its time sitting in a tight loop reading input and making transitions between two states.

Third, the limit point in Grammar 2 forces the PPA to perform a pair of stack operations for every handle of $S \rightarrow a\ S\ b$ it sees. The PPA still behaves linearly, but with a greater overhead than the PPA for Grammar 1 had.

Grammar 3 is an ambiguous grammar used in [23]. It is one of the worst cases for a PPA-based Tomita parser, requiring it to perform numerous stack operations on multiple stacks. The interesting feature we see in this set of tests is a crossover point that occurs between input strings of length 95 to 100. We speculate that this may be in part due to our algorithm's use of memory. Looking at the amount of 'system'

time our parser spends in the operating system kernel, it spends no measurable time there until an input length of 60. The amount of system time then stays steady until a length of about 80–85 where it jumps by a factor of 2–5, at length 100 it jumps by a factor of 3. In contrast, the public domain parser's system time tends to ramp up gradually with the input length. Since there is no I/O overhead for these short inputs to speak of, the system time is most likely due to the memory allocator requesting heap pages from the kernel. That being the case, if the memory usage of the PPA-based Tomita parser is tuned, it may improve its performance on this grammar. To test this hypothesis, we modified both parsers so that all memory was preallocated, after startup, no further requests to the operating system for heap space were made. For a fair comparison, we preallocated the same amount for both parsers, and made no further memory optimizations in the PPA-based parser. The preallocation results for parsing Grammar 3 are shown in Figure 5.12.

In addition, profiling of our parser has shown that over 40% of total run time can be spent doing memory allocation and deallocation when parsing ambiguous grammars. Figure 5.13 shows the results obtained for Grammar 3 when we added a custom-built memory allocator to our parser.

Grammar 4, another ambiguous grammar, is derived from one in [23]. For reasons discussed in Section 7.1.3, reductions in ambiguous grammars by rules with longer

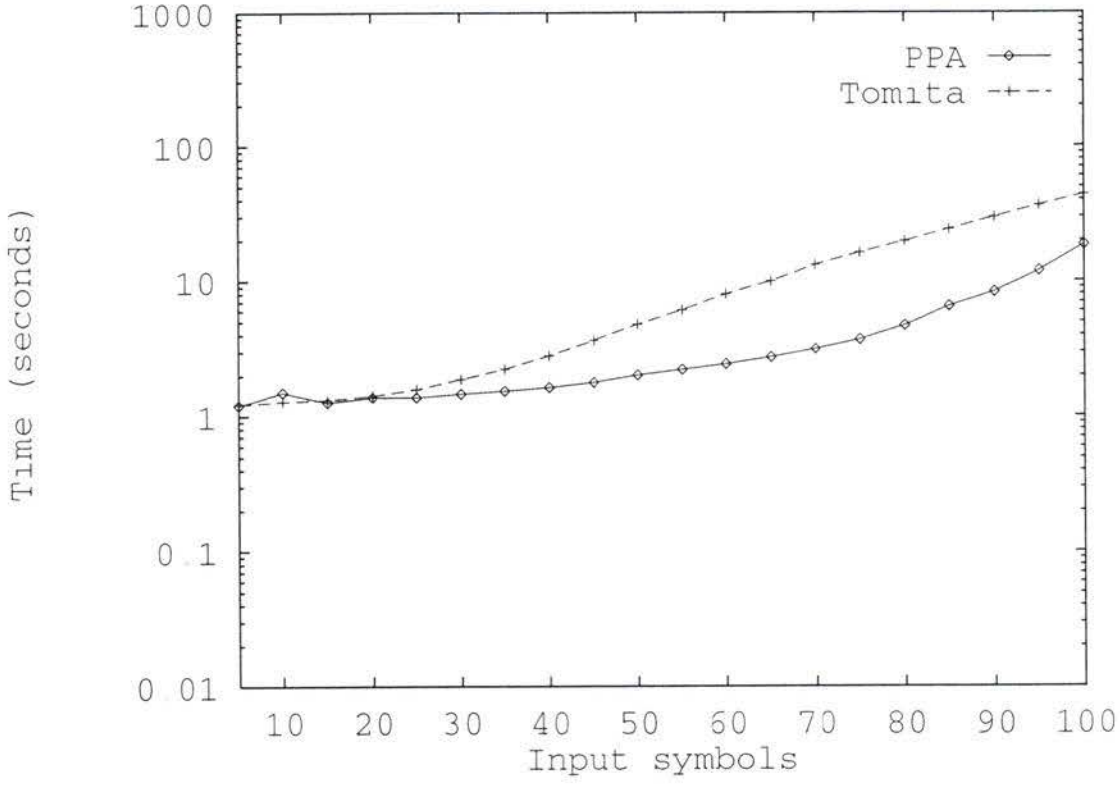


Figure 5.12 Preallocation timings for Grammar 3

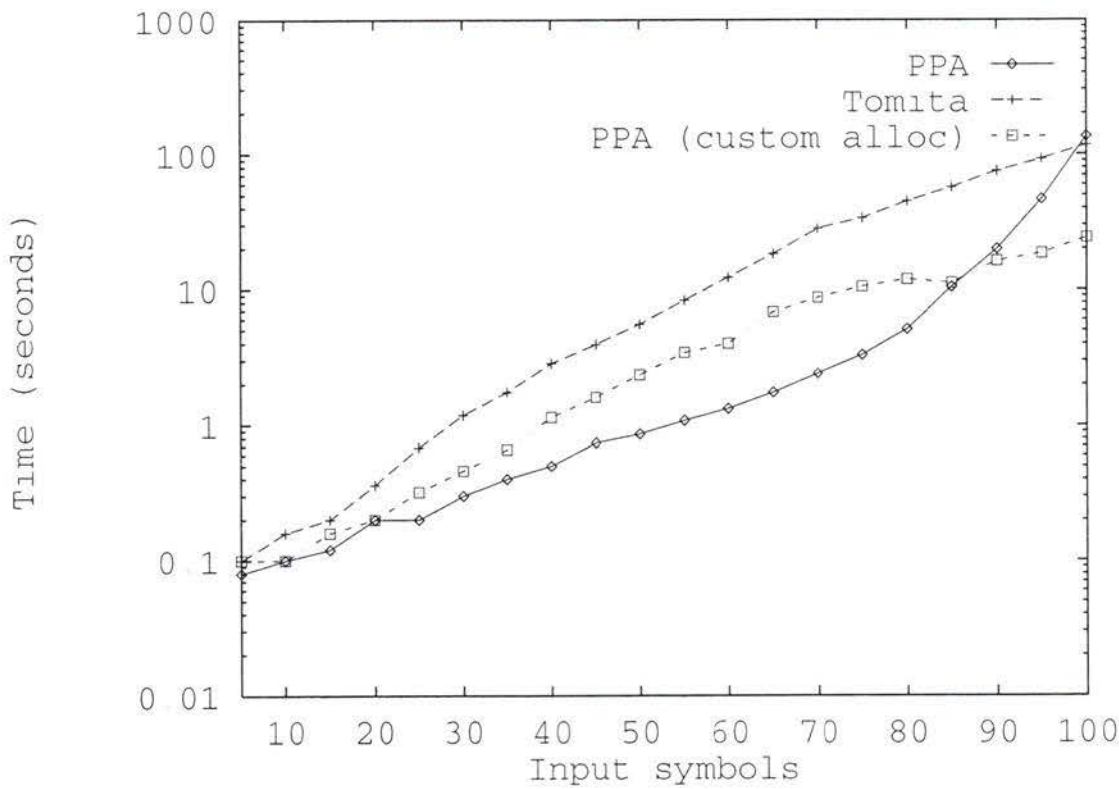


Figure 5 13 Custom allocation timings for Grammar 3

and longer right-hand sides are exponentially more expensive to parse. On the other hand, a PPA always takes negligible time for reductions, as reflected in the results.

Grammar 5 is included to show the effect of grammar expansion on parsing time: the case where $n = 0$ is the same as Grammar 3's PPA. As might be expected, more grammar expansion — and therefore fewer stack operations and shallower stacks — translates into faster parsing times. The $n > 0$ expansions require less memory, and their timing curves are substantially different than the one for $n = 0$, tending to support the hypothesis that memory usage patterns may influence the PPA result for Grammar 3.

Chapter 6

Implementation

6.1 PPA Generation

The code to generate PPAs is comprised of approximately 1400 lines of Python [30], an object-oriented scripting language. The program is divided into a front end and a back end, as follows:

1. The front end takes as input a file containing a CFG and builds the PPA for the grammar. It then "pickles" the result by storing the PPA object and associated information into a file for later consumption.

2. A back end reads the pickled PPA and processes it in some way. Currently there are three different back end programs: `print`, which prints the PPA, `stats`,

which generates statistics about the PPA (e.g. the number of states), `table`, which outputs the PPA as a set of tables suitable for inclusion in a table-driven parser.

In the present implementation, the user is responsible for setting limit points in the grammar. However, the program *does* compute a FAS and thereby is able to suggest a set of limit points; a PPA is not generated unless the set of left contexts is finite.

The front end is capable of performing a connectivity check on the intermediate PFAs that are built. In other words, it can verify that every PFA state is reachable from the start state and that each state has a path to the final state. For large PPAs, this check takes a great deal of time and rarely finds useless states, so it is usually disabled. We conjecture that the presence of useless states in the PPA results from a flaw in the input grammar.

6.2 PPA-Driven Recognition

We have implemented three recognizers, all of which use the same tables as generated by the `table` back end.

1. `lr0`, a LR(0) recognizer using the algorithm from Section 4.8.

2 `tree`, a Tomita recognizer with a tree-structured stack — common stack prefixes are preserved during parsing. `Tree` was a prototype for `graph`, below.

3 `graph`, a Tomita recognizer with a graph-structured stack. This implements the algorithm from Section 5.3, and is the program used in the timing tests of Section 5.4. It uses a reference-counting garbage collection scheme [21] to reclaim unreferenced nodes in the graph-structured stack.

All the recognizers are written in C, and together consist of approximately 1000 lines of code (excluding the PPA tables).

The PPAs generated for most grammars tend to be extremely sparse, and have a large number of states. To conserve memory at parse time, we use an adjacency-list representation for the PPA tables [6]. These adjacency lists are stored consecutively in an array. Another array, indexed by PPA state number, holds pointers to each PPA state's list. If a PPA has v vertices and e edges in its transition diagram, then it has $O(ve)$ space requirements.

Chapter 7

Conclusions and Future Work

7.1 Future Work

7.1.1 Adding Lookahead

To again quote Grune and Jacobs [12, page 205]

> 'Our initial enthusiasm about the clever and efficient LR(0) parsing technique will soon be damped considerably when we find out that very few grammars are in fact LR(0) '

Consider the grammar in Figure 7 1, its PPA is shown in Figure 7 2 The grammar is not LR(0), which manifests itself as the reduce/reduce conflict at the PPA state marked "X " By looking at the PPA, one can see that if one lookahead symbol were

0 S′ → S $
1 S → A a
2 S → B b
3 A → a
4 B → a

Figure 7.1 A non-LR(0) grammar

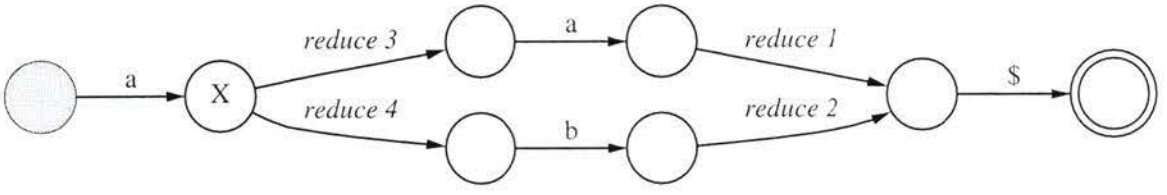


Figure 7.2 PPA for a non-LR(0) grammar

used at "X," it would be easy for the PPA to decide which *reduce* edge to traverse.

As we have presented and implemented them, PPAs take no lookahead into account and are therefore LR(0) parsers. However, they can be changed into LR(1) parsers with relatively little effort. Lookahead symbols are only needed on edges which are not already labeled with a terminal symbol: *reduce*, *push*, and *pop*.

The basic idea is as follows. Start at the end of the edge e which needs a set of lookahead symbols. From that point in the PPA, follow all paths until an edge is found labeled with a terminal symbol (this includes already-computed lookahead symbols adorning edges too). The collection of all such terminal symbols constitutes the set of lookahead symbols for e.

In computing lookahead sets for a PPA, *pop* edges add a slight wrinkle. This is because at run time, traversing a *pop* edge causes the PPA to go to one of a set of states. Fortunately, the set of states T a *pop* can go to is determinable at parser build time. This leaves several approaches as to how to find the lookahead across a *pop* edge.

1. Blind luck. As a special case, some grammars will have PPAs where the lookahead set for a *pop* edge is the same no matter which state in T is returned to. This case can be checked for at parser build time.

2. Almost LR(1). At parser build time, an inexact lookahead set can be computed by taking the union of all lookahead sets for states in T. Although further study is needed, this may be suitable for most practical grammars.

3. Full LR(1). At run time, the exact lookahead set for a *pop* edge can be known if *push* actions place the appropriate lookahead set onto the stack along with the state they push. Of course, this method would incur a slight run-time penalty.

Given that the primary application of our work is Tomita's algorithm, a LR(1) PPA may seem pointless. After all, Tomita's algorithm is designed to deal with LR conflicts! If even a single lookahead is used, however, it reduces the amount of work a Tomita parser must do – there are fewer dead-end paths to follow during parsing.

A PPA may be further extended to be LR(k) by annotating edges with k symbols of lookahead [17], but this is not necessary in most cases. The number of lookahead symbols is a tradeoff between space (to store $k > 0$ lookahead symbols) and the class of grammar that can be recognized. LR(1) is a good choice because it only requires a small amount of extra space over LR(0), yet the LR(1) class of languages encompasses most "important" ones such as programming languages. In addition, lookahead analysis for a PPA would become more expensive for more than one lookahead.

7.1.2 Recognition vs. Parsing

There is a difference between recognition and parsing. A recognizer reads an input string and will simply say "yes" or "no," depending on whether or not the input is in the language that the recognizer accepts. A parser also does the job of a recognizer, but in addition will output at least one derivation if given a valid input string [2].

Our work has primarily focused on recognition. While PPAs know when reductions are performed, they make no attempt to record this information. In the case of unambiguous grammars, there can be only one derivation for a valid input string, and many LR parsers defer the work of remembering this derivation to the user. By permitting semantic actions to be attached to reductions, a LR parser allows the user to supply code to store all or part of a derivation (as needed). Extending the PPA

model to include semantic actions would be straightforward

Ambiguous grammars pose a greater challenge Here an input string may have multiple derivations Tomita [38, 39, 40] solved the problem of recording these derivations by "sub-tree sharing" Basically, Tomita took advantage of the node sharing in the graph-structured stack to construct a directed acyclic graph (DAG) on the fly, this DAG represents all possible derivations of the input It is presently unclear if PPAs, with their different method of using the stack, can employ a similar technique

7 1.3 Even Faster Tomita Parsing

We see two approaches to building even faster PPA-based Tomita parsers

1 Our implementation of the parser extensively uses dynamic memory allocation Study of the parser's memory usage patterns, along with the tuning of our custom-built memory allocator, should further decrease our parser's run time

2 The worst-case performance bound for Tomita's original algorithm is known to be $O(n^{p+1})$, where p is the length of the longest right-hand side of any rule in the grammar [23] This bound was reduced to $O(n^3)$ with a reformulation of Tomita's algorithm by Kipps [23]

A dominant factor in the time complexity of Tomita's original algorithm is

that, upon reduction by a rule $A \rightarrow \alpha$, all paths of length $|\alpha|$ from a stack top have to be found in the graph-structured stack. To obtain his bound of $O(n^3)$, Kipps reduced the amount of work necessary during this step by caching previously-computed paths. In our PPA-based parser, reductions do not require any stack operations. And, when a PPA does need to access the stack, it is only ever accessing the topmost entry in the stack. Therefore, we suspect that by combining our work with Kipps', we may be able to improve the upper time bound of Tomita's algorithm beyond $O(n^3)$.

7.2 Conclusions

Parsing is a key topic in computer science because of its wide variety of application areas. In particular, parsing user input quickly is important because of its high degree of user visibility.

The class of context-free grammars is sufficient to describe many programming languages and parts of natural language. For a large subset of context-free grammars — the LR class — deterministic linear-time methods are known. Parsers for general context-free grammars tend to be slower on unambiguous grammars not in LR, and slower still on ambiguous grammars. Tomita's parsing method was designed for use

in parsing natural languages, and handles ambiguous grammars efficiently

Unfortunately, while much work has been directed at speeding up LR parsers, not much of this is applicable to the more general case of Tomita parsers. In this thesis we have taken steps to remedy this gap in knowledge. Our work has resulted in the speedup of Tomita parsers.

Revisiting early research on LR parsers, we developed an original algorithm to construct PPAs, a variant of pushdown automata. These PPAs recognize more than the handle-finding automata used in standard LR parsers, and as a result perform much fewer stack operations than their counterparts.

Then, we devised a method to combine our PPAs with Tomita's parsers. Our PPA-based Tomita parser typically takes substantially less time to parse than a regular Tomita parser, even for highly ambiguous grammars. In the worst case, an improvement by a factor of ten is shown to occur on most inputs.

Our conclusion is that by trading space for time — a larger LR parser in exchange for faster execution times — we are able to build Tomita parsers which are faster and better suited to more widespread application outside the natural language domain. We believe that further research will make them faster still.

References

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

[2] A V Aho and J D Ullman *The Theory of Parsing, Translation, and Compiling, Volume 1 Parsing* Prentice-Hall, 1972

[3] R C Backhouse An Alternative Approach to the Improvement of $LR(k)$ Parsers *Acta Informatica*, 6 277–296, 1976

[4] A Bhamidipaty and T A Proebsting Very Fast YACC-Compatible Parsers (For Very Little Effort) Technical Report TR 95-09, Department of Computer Science, University of Arizona, 1995

[5] N Chomsky and G A Miller Finite State Languages *Information and Control*, 1 91–112, 1958

[6] T H Cormen, C E Leiserson, and R L Rivest *Introduction to Algorithms* McGraw-Hill, 1990

[7] P J Denning, J B Dennis, and J E Qualitz *Machines, Languages, and Computation* Prentice-Hall, 1978

[8] P Eades, X Lin, and W F Smyth A fast and effective heuristic for the feedback arc set problem *Information Processing Letters*, 47 319–323, 1993

[9] J Earley An Efficient Context-Free Parsing Algorithm *Communications of the ACM*, 13(2) 94–102, 1970

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

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

[12] D Grune and C J H Jacobs *Parsing Techniques A Practical Guide* Ellis Horwood, 1990

[13] S Heilbrunner A Parsing Automata Approach to LR Theory *Theoretical Computer Science*, 15 117–157, 1981

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

[15] R N Horspool Personal communication, October 1997

[16] R N Horspool Personal communication, November 1997

[17] R N Horspool Personal communication, February 1998

[18] R N Horspool and M Whitney Even Faster LR Parsing *Software, Practice and Experience*, 20(6) 515–535, 1990

[19] M Johnson The Computational Complexity of GLR Parsing In Tomita [41], pages 35–42

[20] S C Johnson YACC — Yet Another Compiler Compiler *UNIX Programmer's Manual 7th Edition*, 2B, 1979

[21] R Jones and R Lins *Garbage Collection Algorithms for Automatic Dynamic Memory Management* Wiley, 1996

[22] R M Karp Reducibility Among Combinatorial Problems In R E Miller and J W Thatcher, editors, *Complexity of Computer Calculations*, pages 85–103 Plenum Press, 1972

[23] J R Kipps GLR Parsing in Time $O(n^3)$ In Tomita [41], pages 43–59

[24] D E Knuth On the Translation of Languages from Left to Right *Information and Control*, 8 607–639, 1965

[25] D E Knuth *The Art of Computer Programming, Volume 3 Sorting and Searching* Addison-Wesley, 1973

[26] D C Kozen *Automata and Computability* Springer-Verlag, 1997

[27] F E J Kruseman Aretz On a Recursive Ascent Parser *Information Processing Letters*, 29 201–206, 1988

[28] R Leermakers Recursive ascent parsing from Earley to Marcus *Theoretical Computer Science*, 104 299–312, 1992

[29] H R Lewis and C H Papadimitriou *Elements of the Theory of Computation* Prentice-Hall, 1981

[30] M Lutz *Programming Python* O'Reilly & Associates, 1996

[31] P McLean and R N Horspool A Faster Earley Parser In *Proceedings of the International Conference on Compiler Construction (CC '96)*, pages 281–293, 1996

[32] T J Pennello Very Fast LR Parsing In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, volume 21(7) of *ACM SIGPLAN Notices*, pages 145–151, 1986

[33] F C N Pereira and R N Wright Finite-State Approximation of Phrase-Structure Grammars In E Roche and Y Schabes, editors, *Finite-State Language Processing*, pages 149–173 MIT Press, 1997

[34] P Pfahler Optimizing Directly Executable LR Parsers In *Compiler Compilers, Third International Workshop, CC '90*, pages 179–192 Springer-Verlag 1990

[35] G H Roberts Recursive Ascent An LR Analog to Recursive Descent *ACM SIGPLAN Notices*, 23(8) 23–29, 1988

[36] G H Roberts Another Note on Recursive Ascent *Information Processing Letters*, 32 263–266, 1989

[37] E Speckenmeyer On Feedback Problems in Digraphs In *Graph-Theoretic Concepts in Computer Science*, pages 218–231 Springer-Verlag 1989

[38] M Tomita *An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications* PhD thesis, Carnegie-Mellon University, 1985

[39] M Tomita *Efficient Parsing for Natural Language* Kluwer Academic, 1986

[40] M Tomita An Efficient Augmented-Context-Free Parsing Algorithm *Computational Linguistics*, 13(1–2) 31–46, 1987

[41] M. Tomita, editor. *Generalized LR Parsing*. Kluwer Academic, 1991.

[42] F. W. Weingarten. *Translation of Computer Languages*. Holden-Day, 1973.

Vita

Surname	Aycock
Given Names	John Daniel
Place of Birth	Toronto, Ontario, Canada

Educational Institutions Attended

University of Victoria	1996 – 1998
University of Calgary	1988 – 1992
Grant MacEwan Community College	1987

Degrees Awarded

B Sc University of Calgary	1993

Partial Copyright License

I hereby grant the right to lend my thesis 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:

Faster Tomita Parsing

Author: [redacted]

John Daniel Aycock

April 9, 1998