Generalised LR parsing algorithms

Giorgios Robert Economopoulos
Department of Computer Science
Royal Holloway, University of London

August, 2006

Abstract

This thesis concerns the parsing of context-free grammars. A parser is a tool, defined for a specific grammar, that constructs a syntactic representation of an input string and determines if the string is grammatically correct or not. An algorithm that is capable of parsing any context-free grammar is called a generalised (context-free) parser. This thesis is devoted to the theoretical analysis of generalised parsing algorithms. We describe, analyse and compare several algorithms that are based on Knuth's LR parser. This work underpins the design and implementation of the Parser Animation Tool (PAT). We use PAT to evaluate the asymptotic complexity of generalised parsing algorithms and to develop the Binary Right Nulled Generalised LR algorithm − a new cubic worst case parser. We also compare the Right Nullable Generalised LR, Reduction Incorporated Generalised LR, Farshi, Tomita and Earley algorithms using the statistical data collected by PAT. Our study indicates that the overheads associated with some of the parsing algorithms may have significant consequences on their behaviour.

Acknowledgements

My sincere gratitude goes to Elizabeth Scott and Adrian Johnstone. Their support and guidance throughout the course of my PhD has been invaluable. I would also like to thank my examiners, Peter Rounce and Nigel Horspool, for their constructive comments and suggestions that really improved the thesis.

Many friends and colleagues have helped me over the last four years, but special thanks goes to Martin Green, Rob Delicata, Jurgen Vinju, Tijs van der Storm, Magiel Bruntink, Hartwig Bosse and Urtzi Ayesta. Of course none of this would have been possible without the continued support of my parents, Maggie and Costas, and my brother, Chris.

Finally, I would like to thank Valerie for being there for me when I needed her most.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Most programming languages are implemented using almost deterministic context-free grammars and an LALR(1) parser generator such as Yacc, or GNU Bison. This approach to language development has been accepted for a long time and consequently many people believe that research into parsing is 'done'. This view is somewhat surprising given that there are still many important questions that remain unanswered. For example, the following two problems set by Knuth in his seminal paper [Knu65] are still unsolved:

> Are there general parsing methods for which a linear parsing time can be guaranteed for all grammars?
>
> Are there particular grammars for which no conceivable parsing method will be able to find one parse of each string in the language with running time at worst linearly proportional to the length of the string?

Clearly a lot more research needs to be carried out before we can honestly claim that parsing is 'done'. This thesis makes a significant contribution to *generalised* parsing theory, specifically focusing on Tomita's GLR technique and its extensions.

A generalised parser is a parser that is capable of parsing strings for any context-free grammar. Although the first generalised parsing algorithms were published as far back as the 1960s, they have not generally been used in practice due to their relatively high runtime costs. The relentless improvement of computing power and storage, however, has seen interest in generalised parsing techniques resurface.

One of the main reasons behind the increased usage of generalised parsing is the popularity of languages like C++ that are difficult to parse using the standard deterministic techniques. Another is the increasing importance of software re-engineering tools that perform transformations of legacy software. Often this type of software is written in a programming language whose grammar is ambiguous.

Despite this increase in use, generalised parsers are still relatively inefficient; the most efficient generalised parsing algorithm to date displays $O(n^{2.376})$ complexity. Although it is believed that linear time generalised parsers are unlikely to exist, work should continue on improving the efficiency of generalised parsers. Unfortunately, the existing literature lacks a comprehensive, comparative analysis of generalised parsing techniques. This hinders the understanding of existing techniques and hampers the development of new approaches.

1.2 Contributions

The main contributions of this thesis are the description of the new Binary Right Nulled GLR (BRNGLR) parsing algorithm, the comparative analysis of existing GLR techniques and the development of the Parser Animation Tool (PAT).

PAT is a Java application that graphically displays and animates the operation of GLR parsing algorithms and collects statistical data that abstracts their performance. I implemented six different generalised parsing algorithms and several variants of each approach. The implementations closely follow the algorithms' theoretical description in the thesis. I used PAT to collect statistical data for each of the implemented algorithms and compiled a comparative analysis between the different approaches. This analysis indicates that the performance of parsing techniques adopted by several existing tools can be significantly improved by incorporating simple modifications described in this thesis.

The BRNGLR algorithm is a new parser that displays cubic worst case complexity. Unlike other approaches that achieve cubic complexity the BRNGLR algorithm does not require its grammars to be modified and it constructs a representation of all derivations (during a parse) in at most cubic space. Although the initial idea of the algorithm is due to Scott, I was heavily involved throughout the development of the algorithm. In particular, I developed the prototype of BRNGLR in PAT which was used to analyse the algorithms behaviour and test several optimisations.

In summary, the main contributions of this thesis are as follows: the presentation of the BRNGLR algorithm; the comprehensive analysis and description of existing GLR algorithms; new results which show that the techniques adopted by several existing tools can be significantly improved; and the Parser Animation Tool which provides a way of repeating and understanding the experiments and algorithms presented in this thesis.

Several of the results developed in this thesis have been subsequently published. The relevant papers are listed in the bibliography [JSE04a, JSE04b, JSE04c, JSE].

1.3 Outline of thesis

The thesis is split into four parts. Part I is made up of three chapters and provides the reader with the theory that is required in the rest of the thesis. Chapter 2 focuses on theory related to the specification and parsing of computer languages. Chapter 3 paints a picture of the major developments in generalised parsing and discusses relations between the various techniques. Chapter 4 introduces Tomita's GLR parsing algorithm and Farshi's later correction. A detailed discussion is given of both approaches highlighting some of their drawbacks.

Part II introduces three recent algorithms that use a variety of techniques to improve on the efficiency of the traditional GLR parsing algorithm. Chapter 5 describes the RNGLR algorithm that corrects Tomita's original algorithm by redefining the reduce action in the parse table. In addition to recasting the algorithm in terms of Rekers' SPPF representation, some modifications are proposed which improve its efficiency. Chapter 6 introduces the BRNGLR algorithm; a cubic-time parser based on the RNGLR algorithm that does not require any modification to be made to the grammar. Chapter 7 presents the RIGLR algorithm that attempts to improve the efficiency of a GLR algorithm by reducing the amount of stack activity. Chapter 8 discusses other work that has contributed to the development of generalised parsing, and relates them to the techniques described in this thesis.

Part III contains two chapters. Chapter 9 discusses some of the major applications of GLR and backtracking parsing techniques. As part of the work for this thesis, the algorithms discussed have been implemented in the Parser Animation Tool (PAT). This tool has been used to investigate the practical and theoretical strengths and weaknesses of the algorithms. In parallel, the GTB tool has also been developed. Together, PAT and GTB have been used to run the algorithms on grammars for Pascal, C, Cobol and various smaller test grammars. The results are presented and discussed in Chapter 10.

Part IV contains one chapter that concludes the thesis and maps out possible future work.

Part I

Background material

Chapter 2

Recognition and parsing

In order to execute programs, computers need to be able to analyse and translate programming languages. In general, programming languages are simpler than human languages, but many of the principles involved in studying them are similar.

The compilation process as a whole has been meticulously studied for a long time. However, writing a correct and efficient parser by hand is still a difficult task. Although tools exist that can automatically generate parsers from a grammar specification, limitations with the techniques mean that they often fail.

The early high level programming languages, like FORTRAN, were developed with expressibility rather than parsing efficiency in mind. Some language constructs turned out to be difficult to parse efficiently, but without a proper understanding of the reasons behind this inefficiency, a solution was difficult to come by. The subsequent work on formal language theory by Chomsky provided a solid theoretical base for language development.

The field of research that investigates languages, grammars and parsing is called formal language theory. This chapter provides a brief overview of language classification as defined by Chomsky [Cho56] and introduces the formal definitions and basic concepts that underpin programming language parsing. We discuss general top-down and bottom-up parsing and the standard deterministic LR parsing technique.

2.1 Languages and grammars

In formal language theory a language is considered to be a set of sentences. A *sentence* consists of a sequence of words that are concatenations of a number of symbols from a finite *alphabet*. A finite language can be defined by providing an enumeration of the sentences it contains. However, many languages are infinite and attempting to enumerate their contents would be a fruitless exercise. A more useful approach is to provide a set of rules that describe the structure, or *syntax*, of the language. These rules, collectively known as a *grammar*, can then be used to generate or recognise syntactically correct sentences.

The work pioneered by Chomsky in the late 1950's formalised the notion of a grammar as a generative device of a language[1]. He developed four types of grammars and classified them by the structures they are able to define. This famous categorisation, known as the Chomsky hierarchy, is shown in Table 2.1. Each class of language can be specified by a particular type of grammar and recognised by a particular type of formal machine. The automata in the right hand column of the table are the machines that accept the strings of the associated language.

Grammars	Languages	Automata
Type-0 (Unrestricted)	Recursively enumerable	Turing machine
Type-1 (Context-sensitive)	Context-sensitive	Linear bounded non-deterministic Turing machine
Type-2 (Context-free)	Context-free	Non-deterministic push down automaton
Type-3 (Regular)	Regular	Finite automaton

Table 2.1: The Chomsky hierarchy

The grammars towards the bottom of the hierarchy define the languages with the simplest structure. The rules of a regular grammar are severely restricted and as a result, are only capable of defining simple languages. For example, they cannot generate languages in which parentheses must occur in matched pairs. Despite the limited nature of the regular languages, the corresponding finite automata play an important role in the recognition of the more powerful context-free languages.

The languages defined by context-free grammars are less confined by the restrictions imposed upon their grammar rules. In addition to expressing all the regular languages, they can also define recursively nested structures, common in many programming languages. As a result, the context-free grammars are often used to define the structure of modern programming languages and their automata form the basis of many classes of corresponding recognition and parsing tools.

Certain structural dependencies cannot be expressed by a context-free grammar alone. The context-sensitive grammars define the languages whose structure depends upon their surrounding context. Although attempts have been made to use the context-sensitive grammars to define natural languages, see for example [Woo70], the resulting grammars tend to be very difficult to understand and use. In a similar way, although the recursively enumerable languages, generated by the type-0 grammars, are the most powerful in the hierarchy, their complicated structure means that they are usually only of theoretical interest.

[1] Although unknown to Western world until recently, Pānini, an Indian scholar, produced a formal grammar for Sanskrit between 350BC and 250BC [Sik97].

2.2 Context-free grammars

At the core of a context-free grammar is a finite set of rules. Each rule defines a set of sequences of symbols that it can generate. There are two types of symbols in a context-free grammar; the *terminals*, which are the symbols, letters or words, of the language and the *non-terminals* which can be thought of as variables – they constitute the left hand sides of the rules. An example of a context-free grammar for a subset of the English language is given in Grammar 2.1. The non-terminals are shown as the upper case words and the terminals as the lower case words.

$$
\begin{aligned}
&\textsc{Sentence} \rightarrow \textsc{NounPhrase VerbPhrase}\\
&\textsc{NounPhrase} \rightarrow \textsc{NounPhrase PrepositionalPhrase}\\
&\textsc{NounPhrase} \rightarrow \textsc{Determiner Noun}\\
&\textsc{NounPhrase} \rightarrow \textsc{Noun}\\
&\textsc{VerbPhrase} \rightarrow \textsc{Verb NounPhrase}\\
&\textsc{PrepositionalPhrase} \rightarrow \textsc{Preposition NounPhrase}\\
&\textsc{Noun} \rightarrow I\\
&\textsc{Noun} \rightarrow man\\
&\textsc{Noun} \rightarrow bat\\
&\textsc{Verb} \rightarrow hit\\
&\textsc{Preposition} \rightarrow with\\
&\textsc{Determiner} \rightarrow the
\end{aligned}
\tag{2.1}
$$

2.2.1 Generating sentences

We generate strings from other strings using the grammar rules by taking a string and replacing a non-terminal with the right hand side of one of its rules. So for example, from the string

$$\textsc{NounPhrase VerbPhrase}$$

we can generate

$$\textsc{NounPhrase Verb NounPhrase}$$

by replacing the non-terminal $\textsc{VerbPhrase}$. We often write this as

$$\textsc{NounPhrase VerbPhrase} \Rightarrow \textsc{NounPhrase Verb NounPhrase}$$

and call it a derivation step.

We are interested in the set $L(A)$ of all strings, that contain only terminals, that can be generated from a non-terminal A. This is called the language of A. One of the non-terminals will be designated as the start symbol and the language generated by this non-terminal is called the language of the grammar. For example, the sentence

I hit the man with a bat can be generated from the non-terminal SENTENCE in Grammar 2.1 in the following way:

$$\begin{aligned}
\textsc{Sentence} &\Rightarrow \textsc{NounPhrase VerbPhrase}\\
&\Rightarrow \textsc{Noun VerbPhrase}\\
&\Rightarrow \textit{I}\ \textsc{VerbPhrase}\\
&\Rightarrow \textit{I}\ \textsc{Verb NounPhrase}\\
&\Rightarrow \textit{I hit}\ \textsc{NounPhrase}\\
&\Rightarrow \textit{I hit}\ \textsc{NounPhrase PrepositionalPhrase}\\
&\Rightarrow \textit{I hit}\ \textsc{Determiner Noun PrepositionalPhrase}\\
&\Rightarrow \textit{I hit the}\ \textsc{Noun PrepositionalPhrase}\\
&\Rightarrow \textit{I hit the man}\ \textsc{PrepositionalPhrase}\\
&\Rightarrow \textit{I hit the man}\ \textsc{Preposition NounPhrase}\\
&\Rightarrow \textit{I hit the man with}\ \textsc{NounPhrase}\\
&\Rightarrow \textit{I hit the man with}\ \textsc{Determiner Noun}\\
&\Rightarrow \textit{I hit the man with the}\ \textsc{Noun}\\
&\Rightarrow \textit{I hit the man with the bat}
\end{aligned}$$

A sequence of derivation steps is called a derivation.

The basic idea is to use a grammar to define our chosen programming language. Then given some source code we need to determine if it is a valid program, ie a sentence in the language of the grammar. This can be achieved by finding a derivation of the program. The construction of derivations is referred to as parsing, or syntax analysis.

2.2.2 Backus Naur form

At about the same time as Chomsky was investigating the use of formal grammars to capture the key properties of human languages, similar ideas were being used to describe the syntax of the Algol programming language [BWvW^{+}60]. Backus Naur Form (also known as Backus Normal Form or BNF) is the notation developed to describe Algol and it was later shown that languages defined by BNF are equivalent to context-free grammars [GR62]. Because of its more convenient notation, BNF has continued to be used to define programming languages.

In BNF the non-terminals that declare a rule are separated from the body of the rule by ::= instead of the → symbol. In addition to this, a rule can define a choice of sequences with the use of the | symbol. So for example, Grammar 2.2 is the BNF representation of Grammar 2.1.

$$\begin{aligned}
\textsc{Sentence} &::= \textsc{NounPhrase VerbPhrase} \\
\textsc{NounPhrase} &::= \textsc{NounPhrase PrepositionalPhrase} \mid \\
& \textsc{Determiner Noun} \mid \\
& \textsc{Noun} \\
\textsc{VerbPhrase} &::= \textsc{Verb NounPhrase} \\
\textsc{PrepositionalPhrase} &::= \textsc{Preposition NounPhrase} \\
\textsc{Noun} &::= \mathit{I} \mid \mathit{man} \mid \mathit{bat} \\
\textsc{Verb} &::= \mathit{hit} \\
\textsc{Preposition} &::= \mathit{with} \\
\textsc{Determiner} &::= \mathit{the}
\end{aligned} \tag{2.2}$$

We shall use the BNF notation to define all of the context-free grammars in the remainder of this thesis. In addition to this, we shall maintain the following convention when discussing context-free grammars: lower case characters near the front of the alphabet, digits and symbols like $+$, represent terminals; upper case characters near the front of the alphabet represent non-terminals; lower case characters near the end of the alphabet represent strings of terminals; upper case characters near the end of the alphabet represent strings of either terminals or non-terminals; lower case Greek characters represent strings of terminals and/or non-terminals.

We define a context-free grammar formally as a 4-tuple $\langle \mathbf{N}, \mathbf{T}, \mathbf{S}, \mathbf{P} \rangle$, where $\mathbf{N}$ and $\mathbf{T}$ are disjoint finite sets of grammar symbols, called non-terminals and terminals respectively; $\mathbf{S} \in \mathbf{N}$ is the special start symbol; $\mathbf{P}$ is the finite set of production rules of the form $A ::= \beta$, where $A \in \mathbf{N}$ and β is the string of symbols from $(\mathbf{N} \cup \mathbf{T})^*$. The production rule $S ::= \alpha$ where $S \in \mathbf{S}$ is called the grammar's *start symbol.* We may augment a grammar with the new non-terminal, S', that does not appear on the right hand side of any other production rule. The empty string is represented by the ϵ symbol. For example consider Grammar 2.3 which defines the language $\{ac, abc\}$.

$$\begin{aligned}
S' &::= S \\
S &::= aBc \\
B &::= b \mid \epsilon
\end{aligned} \tag{2.3}$$

The replacement of a single non-terminal in a sequence of terminals and non-terminals is called a *derivation step* and is represented by the $\Rightarrow$ symbol. The application of a number of derivation steps is called a *derivation.* We use the $\overset{*}{\Rightarrow}$ symbol to represent a derivation consisting of zero or more steps and the $\overset{+}{\Rightarrow}$ symbol for derivations of one or more steps. Any string α such that $S \overset{*}{\Rightarrow} \alpha$ is called a *sentential form* and a sentential form that contains only terminals is called a *sentence.* A non-terminal that derives the empty string is called *nullable.* If $A ::= \epsilon$ then $\alpha A \beta \Rightarrow \alpha\beta$.

We call rules of the form $A ::= \alpha\beta$ where $\beta \overset{*}{\Rightarrow} \epsilon$ right nullable rules. A grammar that contains a non-terminal A, such that $A \overset{+}{\Rightarrow} \alpha A\beta$, where $\alpha, \beta \neq \epsilon$, is said to contain self-embedding.

Since there is often a choice of non-terminals to replace in each derivation step, one of two approaches is usually taken; either the leftmost or the rightmost non-terminal is always replaced. The former approach achieves a *leftmost* derivation whereas the latter produces a *rightmost* derivation. The derivation in Section 2.2 is a leftmost derivation.

2.2.3 Recognition and parsing

An important aspect of the study of context-free grammars is the recognition of the sentences of their languages. It turns out that given any context-free grammar there is a Push Down Automaton (PDA) that accepts precisely the language of the grammar. Tools that take a string and determine whether or not it is a sentence are called *recognisers.*

In addition to implementing a recogniser, many applications that use context-free grammars also want to know the syntactic structure of any strings that are recognised. Since the rules of a grammar reflect the syntactic structure of a language, we can build up a syntactic representation of a recognised string by recording the rules used in a derivation. Tools that construct some form of syntactic representation of a string are called *parsers.*

2.2.4 Parse trees

A useful way of representing the structure of a derivation is with a *parse tree.* A parse tree is a rooted tree with terminal symbols of the sentence appearing as leaf nodes and non-terminals as the interior nodes. If an interior node is labelled with the non-terminal A then its children are labelled $A_1, ..., A_j$, where the rule $A ::= A_1...A_j$ has been used at the corresponding point in the derivation. The *root* of a parse tree is labelled by the start symbol of the associated grammar and its *yield* is defined to be the sequence of terminals that label its leaves. Figure 2.1 is a parse tree representing the derivation of the sentence *I hit the man with a bat* for Grammar 2.1.

2.3 Parsing context-free grammars

The aim of a parser is to determine if it is possible to derive a sentence from a context-free grammar whilst also building a representation of the grammatical structure of the sentence. There are two common approaches to building a parse tree, top-down and bottom-up. This section presents the top-down and bottom-up parsing approaches.

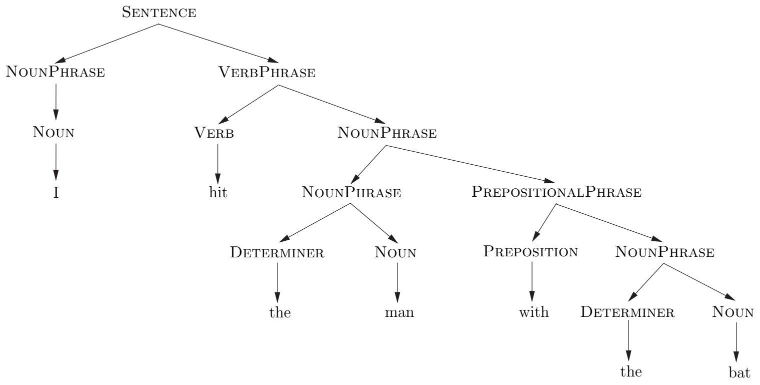


Figure 2.1: A parse tree of Grammar 2.1 for the string *I hit the man with the bat.*

2.3.1 Top-down parsing

A top-down parser attempts to derive a sentence by performing a sequence of derivation steps from the start rule of the grammar. It gets its name from the order in which the nodes in the parse tree are constructed during the parsing process; each node is created before its children.

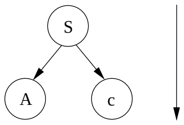
Top-down parsers are usually implemented to produce leftmost derivations and are sometimes called predictive parsers because of the way they 'predict' the rules to use in a derivation. To parse a string we begin with the start symbol and predict the right hand side of the start rule. If the first symbol on the right hand side of the predicted rule is a terminal that matches the symbol on the input, then we read the next input symbol and move on to the next symbol of the rule. If it is a non-terminal then we *predict* the right hand side of the rule it defines.

For example consider the following top-down parse of the string *bdac* for Grammar 2.4.

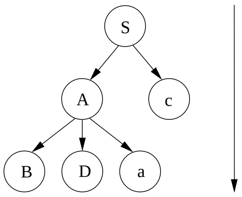
$$\begin{aligned} S &::= Ac \\ A &::= BDa \mid DBa \\ B &::= b \\ D &::= d \end{aligned} \tag{2.4}$$

The start symbol of the grammar is defined to be S, so we begin by creating the root of the parse tree and labeling it S.

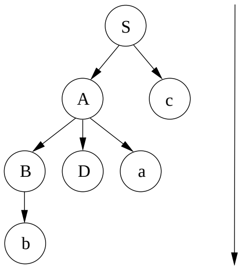
The right hand side of the start rule is made up of two symbols so the new sentential form generated is Ac. We create two new nodes, labelled A and c respectively, and add them as children of the root node.



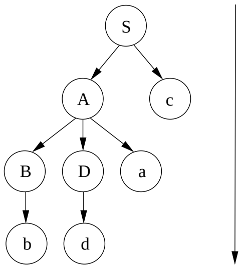
At this point we are looking at the first symbol of the sentential form Ac. Since A is a non-terminal, we need to predict the right hand side of the rule it defines. However, the production rule for A has two alternates, BDa and DBa, so we are required to pick one. If the first alternate is selected, then the parse tree shown below is constructed.



The new sentential form is $BDac$, so we continue by predicting the non-terminal B and create the following parse tree.



We then match the symbol b in the sentential form $bDac$ with the next input symbol and continue to predict the non-terminal D. This results in the parse tree below being constructed.



The only symbols in the sentential form that have not yet been parsed are the terminals dac. A successful parse is completed once all of these symbols are matched to the remaining input string.

Perhaps the most popular implementation of the top-down approach to parsing is the recursive descent technique. Recursive descent parsers define a parse function for each production rule of the grammar – when a non-terminal is to be matched, the parse function for that non-terminal is called. As a result recursive descent parsers are relatively straightforward to implement and their structure closely reflects the structure of the grammar.

This approach to parsing can run into problems when there is a choice of alternates to be used in a derivation. For example, in the above parse if we had picked the rule DBa instead of BDa then the parse would have failed.

We could backtrack to the point that a choice was made and choose a different alternative, but this approach can result in exponential costs and in certain cases may not even terminate. A particular problem is caused by grammars with left recursive rules.

Recursive grammars

A grammar has *left (or right) recursion* if it contains a non-terminal A and a derivation $A \overset{+}{\Rightarrow} \alpha A \beta$ where $\alpha \overset{*}{\Rightarrow} \epsilon$ (or $\beta \overset{*}{\Rightarrow} \epsilon$). If $\alpha \neq \epsilon$ (or $\beta \neq \epsilon$) then the recursion is referred to as *hidden*.

Unfortunately, the standard recursive descent parsers, and most parsers that produce leftmost derivations, cannot easily parse left recursive grammars. In the case of a left recursive rule like $A ::= Ab$, the parse function for A will repeatedly call itself without matching any input.

Although left recursion can be mechanically removed from a grammar [AU73], the removal process alters the structure of the grammar and hence the structure of the parse tree.

2.3.2 Bottom-up parsing

A bottom-up parser attempts to build up a derivation in reverse, effectively deriving the start symbol from the string that is parsed. Its name refers to the order in which the nodes of the parse tree are constructed; the leaf nodes at the bottom are created first, followed by the interior nodes and then finally the root of the tree.

It is natural for the implementation of a bottom-up parser to produce a rightmost derivation. A string is parsed by *shifting* (reading) a number of its symbols until a string of symbols is found that matches the right hand side of a rule. The portion of the sentential form that matches the right hand side of a production rule is called a *handle*. Once the handle is found, the substring in the sentential form is *reduced* (replaced) by the non-terminal on the left hand side of the rule. For each of the terminal symbols shifted, a leaf node in the parse tree is constructed. When a reduction is performed, a new intermediate node is created and the nodes labelled by the symbols in the handle are added to it as children. A string is successfully parsed when a node labelled by the grammar's start symbol is created in the parse tree and the sentential form only contains the grammar's start symbol.

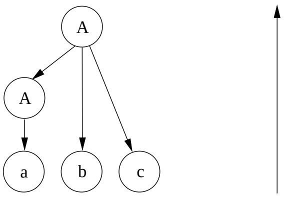
For example, consider Grammar 2.5 and the parse tree constructed for a bottom-up parse of the string *abcd*.

$$\begin{aligned} S &::= Ad \\ A &::= Abc \mid a \end{aligned} \tag{2.5}$$

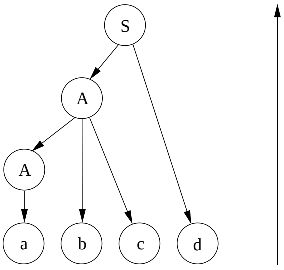
We begin by shifting the first symbol from the input string and create the node labelled a in the parse tree.

Since there is a rule $A ::= a$ in the grammar, we can use it to reduce the sentential form a to A. We create the new intermediate node in the parse tree labelled A and make it the parent of the existing node labelled a.

We continue by shifting the next two input symbols to produce the sentential form *Abc*. At this point we can reduce the whole of the sentential form to A using the rule $A ::= Abc$. We create the new node labelled A as a parent of the nodes labelled A, b and c in the parse tree.



Once the final input symbol is shifted, we can reduce the substring *Ad* in the sentential form to the start symbol S and create the root node of the parse tree to complete the parse.



Bottom-up parsers that are implemented to perform these shift and reduce actions are often referred to as shift-reduce parsers.

To assist our later discussions of the bottom-up parsing technique we shall define the notion of a *viable prefix* and a *viable string*. A viable prefix is a substring of a sentential form that does not continue past the end of the handle. A viable string is a viable prefix that includes a sentential form's handle. We define a handle more formally as a substring γ, that is the right hand side of a grammar rule and that appears in a sentential form β that can be replaced to create another sentential form as part of a derivation.

Although standard bottom-up parsers are often difficult to implement by hand, the development of parser generator tools, like Yacc, have resulted in the bottom-up approach to parsing becoming very popular.

The class of grammars parsable using a standard deterministic bottom-up technique is larger than the class that can be parsed with a standard deterministic recursive descent parser.

2.3.3 Parsing with searching

Recall that during both of our top-down and the bottom-up example parses, we came across a sentential form which had a choice of production rules to predict or reduce by. Fortunately in both cases, we picked a rule that succeeded in producing a derivation. Clearly, parsing may not always be this straightforward.

To deal with such problems a parser can implement one of two obvious approaches. It can either pick the first alternate encountered, as we did in the examples, and record the sentential form that the choice was made at, or parse all possibilities at the same time.

Parsers that adopt the latter approach are the focus of this thesis. We finish this section with a brief discussion of ambiguous grammars. The rest of this chapter is devoted to the standard deterministic LR parsing technique which forms the basis of the general, GLR, parser.

2.3.4 Ambiguous grammars

For some grammars it turns out that certain strings have more than one parse tree. These grammars are called *ambiguous*. For example, consider Grammar 2.6, which defines the syntax of simple arithmetic expressions.

$$\begin{aligned} S' &::= E \\ E &::= E + E \mid E * E \mid a \end{aligned} \tag{2.6}$$

There are two parse trees that can be constructed for the string $a + a * a$ which are shown in Figure 2.2.

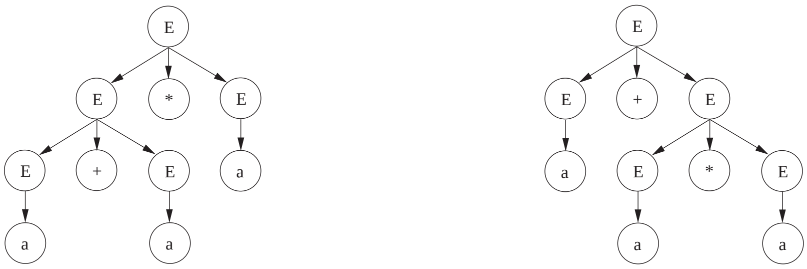


Figure 2.2: Two different parse trees for input $a + a * a$ of Grammar 2.6.

Clearly the searching involved in parsing ambiguous grammars can be expensive. Since context-free grammars are used to define the syntax of programming languages their parsers play an important role in the compilation process of a program, and thus need to be as efficient as possible. As a result several techniques have been developed that are capable of parsing certain non-ambiguous grammars efficiently. The next section focuses on a class of grammars for which an efficient bottom-up parsing technique can be mechanically produced.

2.4 Parsing deterministic context-free grammars

A computer can be thought of as a finite state system and theoreticians often describe what a computer is capable of doing using a Turing machine as a model. Turing machines are the most powerful type of automaton, but the construction and implementation of a particular Turing machine is often infeasible. Other types of automata exist that are useful models for many software and hardware problems. It turns out that the deterministic pushdown automata (DPDA) are capable of modeling a large and useful subset of the context-free grammars. Although these parsers do not work for all context-free grammars, the syntax of most programming languages can be defined by them.

In his seminal paper [Knu65] Donald Knuth presented the LR parsing algorithm as a technique which could parse the class of deterministic context-free grammars in linear time. This section presents that algorithm and shows how the required automata are constructed.

2.4.1 Constructing a handle finding automaton

One of the fundamental problems of any shift-reduce parser is the location of the handle in a sentential form. The approach taken in the previous section compared each sentential form with all the production rules until a handle was found. Clearly this approach is far from ideal.

It turns out that it is possible to construct a Non-deterministic Finite Automaton (NFA) from a context-free grammar to recognise exactly the handles of all possible sentential forms. A finite automaton is a machine that uses a transition function to move through a set of states given a string of symbols. One state is defined to be the automaton's start state and at least one state is defined as an accept state.

We now give a brief description of an NFA for a grammar. Full details can be found in texts such as [ASU86]. Our description here is based on that given in [GJ90].

Finite automata are often represented as transition diagrams, where the states are depicted as nodes and the transition function is defined by the labelled edges between the states. We can construct a finite automaton that accepts the handles of a context-free grammar by labeling the edges with grammar symbols and using the states to represent the substring of a production rule that has been recognised in a derivation. Each of the states are labelled by an *item* − a production rule of the form $(A ::= \alpha \cdot \beta)$ where the substring to the left of the $\cdot$ symbol is a viable prefix of a handle. The accept states of the automaton are labelled by an item of the form $(A ::= \alpha\beta\cdot)$ which indicates that the handle $\alpha\beta$ has been located.

We build the NFA of a context-free grammar by first constructing separate NFA's for each of the grammar's production rules. Given a rule $A ::= \alpha\beta$ we create the start state of the automaton labelled by the item $(A ::= \cdot\alpha\beta)$. For each state that is labelled by an item of the form $(A ::= \alpha \cdot x\beta)$, we create a new state labelled $(A ::= \alpha x \cdot \beta)$ and add an edge, labelled x, between the two states.

These separate automata recognise all right hand sides of a grammar's production rules. Although this includes the handles of any sentential form, they also recognise the substrings that are not handles. We can ensure that only handles are recognised by only considering the right hand sides of rules that can be derived from the start rule of the grammar. To achieve this, the automata are joined by ϵ transitions from the states that are labelled with items of the form $(A ::= \alpha \cdot B\beta)$, to the start state of the automaton for B's production rule. The start state of this new combined automaton is the state labelled by the item $(S' ::= \cdot S)$.

For example, the NFA for Grammar 2.7 is shown in Figure 2.3.

$$\begin{aligned} &S' ::= S \\ &S ::= aAc \mid bAdd \\ &A ::= b \end{aligned} \tag{2.7}$$

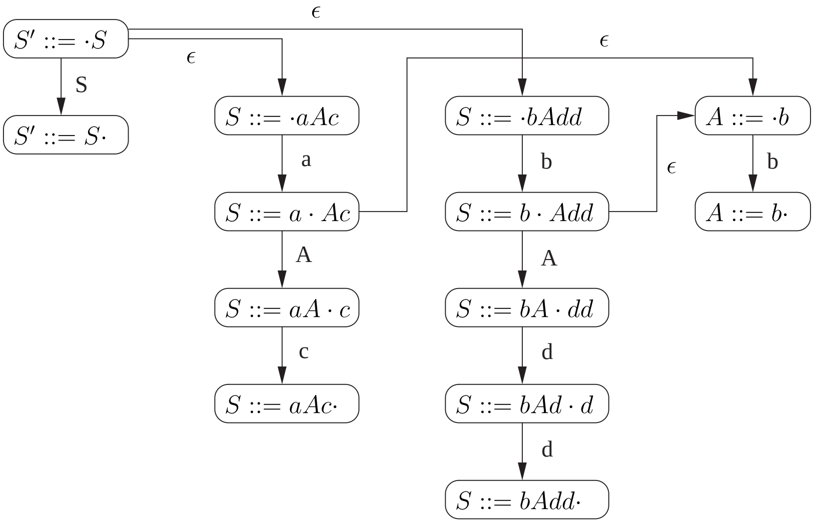


Figure 2.3: The NFA of Grammar 2.7.

To recognise a handle of a sentential form we traverse a path from the start state of the NFA labelled by the symbols in the sentential form. If we end up in an accept state, then we have found the left-most handle of the sentential form.

The traversal is complicated by the fact that the automaton is non-deterministic. For now we take the straightforward approach and follow all traversals in a breadth-first manner. For example, consider the sentential form abc and the NFA in Figure 2.3. We begin by traversing the two ϵ-edges from the start state. The states we reach are labelled $(S ::= \cdot aAc)$ and $(S ::= \cdot bAdd)$. We read the first symbol in the sentential form and traverse the edge to state $(S ::= a \cdot Ac)$. Because we cannot traverse the edge labelled b from state $(S ::= \cdot bAdd)$ we abandon that traversal.

From the current state there is one edge labelled A and another labelled ϵ. Since the next symbol is not A we can only traverse the ϵ-edge that leads to the state labelled $(A ::= \cdot b)$. From there we read the b and traverse the edge to state $(A ::= b\cdot)$. Since this state is an accept state, we have successfully found the handle A in the sentential form abc.

Although this approach to finding handles can be used by bottom-up parsers, the non-deterministic nature of the NFA makes the traversal inefficient. However, it turns out that it is possible to convert any NFA to a Deterministic Finite Automaton (DFA) with the use of the subset construction algorithm [AU73].

The subset construction algorithm performs the ϵ-closure from a node, v, to find the set of nodes, W, that can be reached along a path of ϵ-edges in the NFA. A new node, y is constructed in the DFA that is labelled by the items of the nodes in W. Then for the set of nodes, γ, that can be reached by an edge labelled x from a node in W, we create a new node z in the DFA. We label z with the items of the nodes in

γ and the items of the nodes found by performing the ϵ-closure on each node in γ. We begin the subset construction from the start state of the NFA and continue until no new DFA nodes can be created.

The node created by the ϵ-closure on the start state of the NFA becomes the start state of the DFA. The accept states of the DFA are labelled by items of the form $A ::= \beta\cdot$. Performing the subset construction on the NFA in Figure 2.3, we construct the DFA in Figure 2.4.

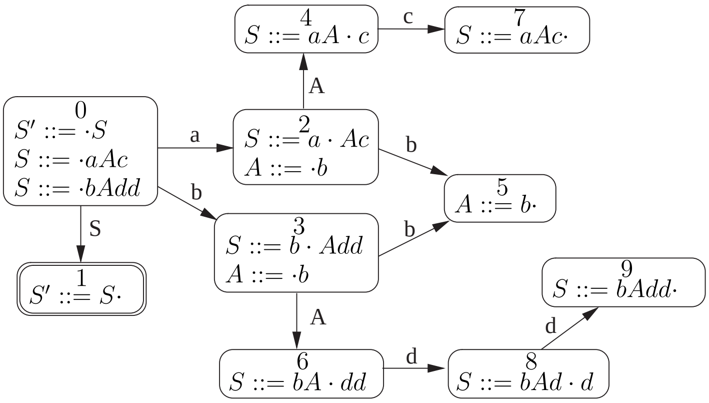


Figure 2.4: The DFA of Grammar 2.7.

A DFA is an NFA which has at most one transition from each state for each symbol and no transitions labelled ϵ. It is customary to label each of the states of a DFA with a distinct *state number* which can then be used to uniquely identify the states. We shall always number the start state of a DFA 0. The state labelled by the item $S' ::= S\cdot$ is the final accepting state of the DFA and is drawn with a double circle.

Parse tables

It is often convenient to represent a DFA as a table where the rows are labelled by the DFA's state numbers and the columns by the symbols used to label the transitions between states. In addition to the terminal and non-terminal symbols that label the transitions of the DFA, the LR parse tables also have a column for the special end-of-string symbol \$.

So as to avoid including the rules used by a reduction in the parse table, we enumerate all of the alternates in the grammar with distinct integers. For example, we label the alternates in Grammar 2.7 in the following way.

0. $S' ::= S$
1. $S ::= aAc$
2. $S ::= bAdd$
3. $A ::= b$

The parse table for Grammar 2.7 is constructed from its associated DFA in the following way. For each of the transitions labelled by a terminal, x, from a state v to a state w, a *shift* action, sw is added to row v, column x. If x is a non-terminal then a *goto* action, gw, is added to entry (v, x) instead. For each state v that contains an item of the form $(A ::= \beta\cdot)$, where N is the item's associated rule number, rN is added to all entries of row v whose columns are labelled by terminals and \$. The accept action *acc* is added to the row labelled by the accept state of the DFA and column \$. The parse table for Grammar 2.7 is shown in Figure 2.2.

	a	b	c	d	$	A	S
0	s2	s3					g1
1					acc		
2		s5				g4	
3		s5				g6	
4			s7				
5	r3	r3	r3	r3	r3		
6				s8			
7	r1	r1	r1	r1	r1		
8				s9			
9	r2	r2	r2	r2	r2		

Table 2.2: The parse table for Grammar 2.7.

Certain grammars generate parse tables with more than one action in a single entry. Such entries are called *conflicts*. There are two types of conflict that can occur: the first, referred to as a *shift/reduce* conflict, contains one shift and one or more reduce actions in an entry; the second, called a *reduce/reduce* conflict, has more than one reduce action in a single state. For example, consider Grammar 2.8 and the DFA shown in Figure 2.5. The associated parse table in Table 2.3 has a reduce/reduce conflict in state 5.

$$
\begin{array}{l}
0.\ S' ::= S \\
1.\ S ::= aBc \\
2.\ S ::= aDd \\
3.\ B ::= b \\
4.\ D ::= b
\end{array}
\tag{2.8}
$$

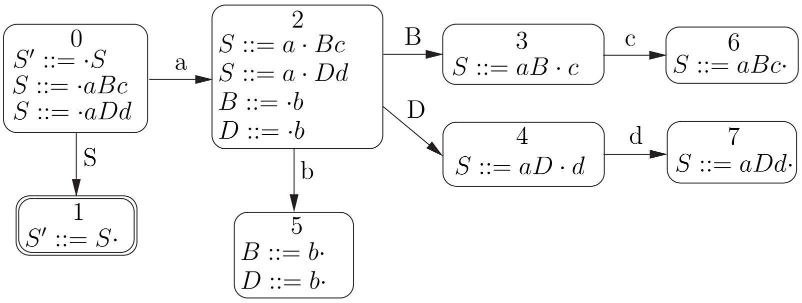


Figure 2.5: The DFA for Grammar 2.8.

	a	b	c	d	\$	B	D	S
0	s2							g1
1					acc			
2		s5				g3	g4	
3			s6					
4				s7				
5	r3/r4	r3/r4	r3/r4	r3/r4	r3/r4			
6	r1	r1	r1	r1	r1			
7	r2	r2	r2	r2	r2			

Table 2.3: The parse table with conflicts for Grammar 2.8.

2.4.2 Parsing with a DFA

We have seen that a DFA can be constructed from an NFA that accepts precisely all handles of a sentential form. It is straightforward to perform a deterministic traversal of this DFA to find a handle of a sentential form. In fact, we extend this traversal to determine if a string is in the language of the associated grammar.

We begin by reading the input string one symbol at a time and performing a traversal through the DFA from its start state. If an accept state is reached for the input consumed, then the leftmost handle of the current sentential form has been located. At this point the parser replaces the string of symbols in the sentential form that match the right hand side of the handle's production rule, with the non-terminal on the left hand side of the production rule. Parsing then resumes with the new sentential form from the start state of the DFA. If an accept state is reached and the start symbol is the only symbol in the sentential form, then the original string is accepted by the parser.

The approach of repeatedly feeding the input string into the DFA is clearly inefficient. The initial portion of the input may be read several times before its handle is found. Consequently a stack is used to record the states reached during a traversal of a given string. When an accept state is reached and a handle γ has been located, the top $|\gamma|$ states are popped off the stack and parsing resumes from the new state at the top of the stack. This prevents the initial portion of the input being repeatedly read.

Next we discuss the LR parsing algorithm that uses a stack to perform a traversal of a DFA.

2.4.3 LR parsing

Knuth's LR parser parses all LR grammars in at most linear time. An LR grammar is defined to be a context-free grammar for which a parse table without any conflicts can be constructed. Strictly speaking there are different forms of LR DFA, LR(0), SLR(1), LALR(1) and LR(1). For the moment we shall not specify which form we are using, the following discussion applies to all of them.

An LR parser works by reading the input string one symbol at a time until it has located the leftmost handle γ of the input. At this point it reduces the handle by popping $|\gamma|$ states off the stack and then pushing the goto state onto the stack. A parse is successful if all the input is consumed and the state on the top of the stack is the DFA's accept state. The formal specification of the algorithm is as follows.

LR algorithm

```
**input data** start state $S_S$, accept state $S_A$, LR table $\mathcal{T}$, input string $a_1...a_n$

push $ and then $S_S$ on to the stack
**for** $i = 0$ to $n$ **do**
    let $l$ be the state on the top of the stack
    **if** $sk \in \mathcal{T}(l, a_{i+1})$ **then**
        push $a_{i+1}$ and then $k$ on to the stack
    **else if** $rk \in \mathcal{T}(l, a_{i+1})$ **then**
        find rule number $k$ such that $A ::= \beta$
        pop $2\times \mid \beta \mid$ symbols off the stack
        let $t$ be the state on the top of the stack
        let $u \in \mathcal{T}(t, A)$
        push $A$ and then $u$ on to the stack
    **else if** $acc \in \mathcal{T}(l, a_{i+1})$ **then**
        **return** *success*
    **else return** *error*
```

To demonstrate the operation of the LR parsing algorithm we trace the stack activity during the parse of the string *abc* with Grammar 2.7 and the parse table shown in Table 2.2. Figure 2.6 shows the contents of the stack after every action is performed in the parse.

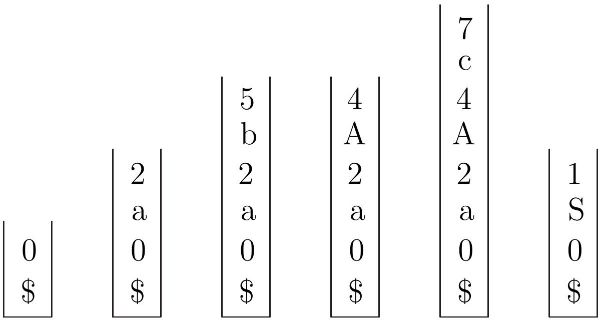


Figure 2.6: The LR parse stacks for the parse of *abc*.

Although it is not strictly necessary to push the recognised symbols onto the stack, we do so here for clarity. However, it is now necessary to pop twice as many elements as the rule's right hand side when a reduction is performed. In order to know when all the input has been consumed the special end-of-string symbol, \$, is added to the end of the input string.

We begin the parse by pushing \$ and the start state of the DFA onto the stack. Since no reduction is possible from state 0, we read the first input symbol, a, and perform the shift to state 2 by pushing the a and then 2 onto the stack. From state 2 we read the next symbol, b, and perform the shift to state 5. State 5 contains a reduction by rule 3, $A ::= b$, so we pop the top two symbols off the stack to reveal state 2. The parse table contains the goto $g4$ in entry $(2, A)$, so we push A and then 4 onto the stack. We continue by pushing the next input symbol, c, and state 7 onto the stack for the action $s7$ in state 4. The reduce by rule 1, $S ::= aAc$, causes the top 6 elements to be popped and S and the goto state 1 to pushed onto the stack. Since the next input symbol is \$ and the parse table contains the *acc* action in entry $(1, \$)$, the string *abc* is accepted by the parser.

2.4.4 Parsing with lookahead

The DFA we have described up to now is called the LR(0) DFA. The LR parsing algorithm requires the parse table to be conflict free, but it is very easy to write a grammar which is not accepted by an LR(0) parser. The problem arises when there is a *conflict* (more than one action) in a parse table entry. For example, consider Grammar 2.8 and the associated parse table shown in Table 2.3.

It turns out that for some grammars these types of conflicts can be resolved with the addition of lookahead symbols to items in the DFA. For the above example, the reduce/reduce conflict can be resolved with the use of just one symbol of lookahead to guide the parse. In the remainder of this section we discuss ways to utilise a single symbol of lookahead.

For a non-terminal A we define a *lookahead set* to be any set of terminals which immediately follow an instance of A in the grammar. A reduction $(A ::= \alpha\cdot)$ is only applicable if the next input symbol, the *lookahead*, appears in the given lookahead set of A. In order to define useful lookahead sets we require the following definitions.

$$\begin{array}{ll} \mathit{first}_{\mathbf{T}}(x) & = \{t \in \mathbf{T} \mid \text{for some string of symbols } \beta,\ x \overset{*}{\Rightarrow} t\beta\} \\ \mathit{first}(\epsilon) & = \{\epsilon\} \\ \mathit{first}(x\gamma) & = \begin{cases} \mathit{first}_{\mathbf{T}}(x) \cup \mathit{first}(\gamma), & \text{if } x \overset{*}{\Rightarrow} \epsilon \\ \mathit{first}_{\mathbf{T}}(x), & \text{otherwise} \end{cases} \end{array}$$

For a non-terminal A we define

$$\mathit{follow}(A) \quad = \{t \mid t \in \mathbf{T} \text{ and } S \overset{*}{\Rightarrow} \beta A t \alpha\}$$

If there is a derivation of the form $S \overset{*}{\Rightarrow} \beta A$ then \$ is also added to $\mathit{follow}(A)$. In particular, $\$ \in \mathit{follow}(S)$ [SJ04].

We now consider two types of LR DFA, SLR(1) and LR(1), that differ only in the lookahead sets that are calculated.

SLR(1)

We call a lookahead set of a non-terminal A *global* when it contains all terminals that immediately follow some instance of A. This is precisely the largest possible lookahead set for A and is exactly the set $\mathit{follow}(A)$. The SLR(1) DFA construction uses these global lookahead sets. A reduction is in row h and column x of the SLR(1) parse table if and only if $(A ::= \alpha\cdot)$ is in h and $x \in \mathit{follow}(A)$.

Consider Grammar 2.8. The LR(0) parse table shown in Table 2.5 contains a reduce/reduce conflict in state 5. If instead we construct the SLR(1) DFA the corresponding parse table has no conflicts as can be seen in Figure 2.7 and Table 2.4.

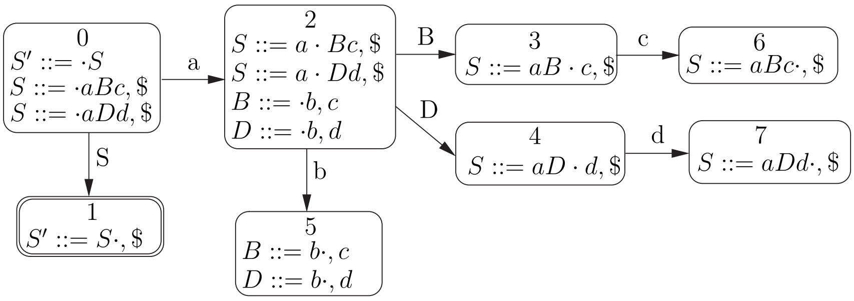


Figure 2.7: The SLR(1) DFA for Grammar 2.8.

	a	b	c	d	\$	B	D	S
0	s2							g1
1					acc			
2		s5				g3	g4	
3			s6					
4				s7				
5			r3	r4				
6					r1			
7					r2			

Table 2.4: The SLR(1) parse table for Grammar 2.8.

However, it is easy to come up with a grammar that does not have an SLR(1) parse table. For example consider Grammar 2.9. The associated SLR(1) DFA contains a reduce/reduce conflict in state 7.

$$\begin{array}{l} 0.\ S' ::= S \\ 1.\ S ::= aBc \\ 2.\ S ::= aDd \\ 3.\ S ::= Bd \\ 4.\ B ::= b \\ 5.\ D ::= b \end{array} \tag{2.9}$$

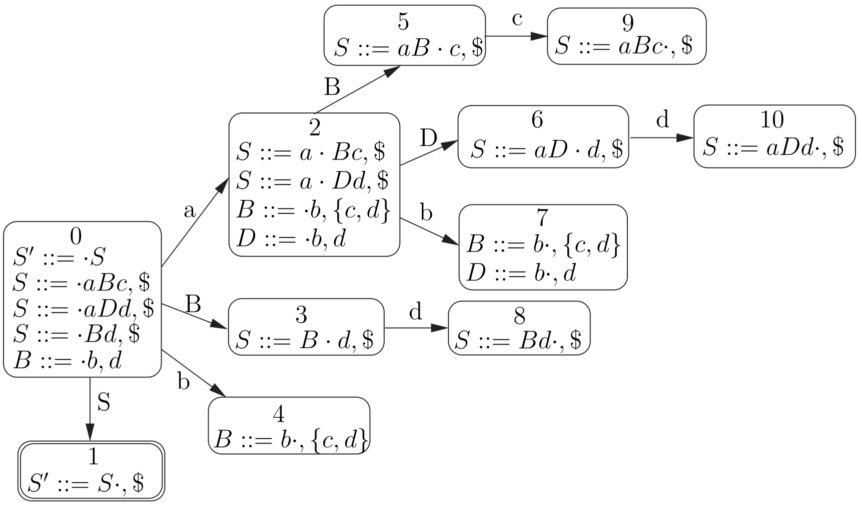


Figure 2.8: The SLR(1) DFA for Grammar 2.9.

	a	b	c	d	\$	B	D	S
0	s2	s4				g3		g1
1					acc			
2		s7				g5	g6	
3				s8				
4			r4	r4				
5			s9					
6				s10				
7			r4	r4/r5				
8					r3			
9					r1			
10					r2			

Table 2.5: The SLR(1) parse table for Grammar 2.9.

To address this we consider the LR(1) DFA's.

LR(1)

Instead of calculating the global lookahead set we can further restrict the applicable reductions by calculating a more restricted lookahead set. We call a lookahead set *local* when it contains the terminals that can immediately follow a particular instance of a non-terminal.

In the LR(1) DFA construction local lookahead sets are used in the following way. For a state containing an item of the form $(B ::= \alpha \cdot A\beta, \gamma)$ the subsequent reduction for a rule defined by A contains the local lookahead set for this instance of A calculated by *first*$(\beta\gamma)$.

For example, the LR(1) DFA of Grammar 2.9 eliminates the reduce/reduce conflict in state 7 of Figure 2.8 and Table 2.5 by restricting the lookahead set of the reduction $(B ::= b\cdot)$ from $\{c, d\}$ to $\{c\}$. This is because the item originates from $(S ::= a \cdot Bc, \$)$ in state 2. The local lookahead set for the instance of B in $S ::= aBc$ is $\{c\}$. (See [ASU86] for full details on LR(1) DFA construction.)

The set of LR(1) grammars strictly includes the SLR(1) grammars. However, in many cases this extra power comes at the price of a potentially large increase in the number of states in the DFA. We discuss the size of the different parse tables in Chapter 10.

The LR(1) DFA construction was defined in Knuth's seminal paper on LR parsing [Knu65], but at the time the LR(1) parse tables were too large to be practical. DeRemer later developed the SLR(1) and LALR(1) DFA's in order to allow more sharing of nodes therefore reducing the size of the DFA [DeR71].

LALR(1)

LALR(1) DFA's are constructed by merging states in the LR(1) DFA whose items only differ by the lookahead set. The LALR(1) DFA for a grammar Γ contains the same number of states but fewer conflicts than the SLR(1) DFA for Γ.

Using k symbols of lookahead

Although it is possible to increase the amount of lookahead that a parser uses, it is not clear that the extra work involved is justified. The automata increase in size very quickly, and for every LR(k) parser it is easy to write a grammar which requires $k+1$ amount of lookahead.

2.5 Summary

This chapter has presented the theory and techniques that are needed to build an LR parser. LR parsers are the most powerful deterministic parsing technique, but despite being used to define the syntax of many modern programming languages, it is easy to write a grammar which is not LR(1). The next chapter paints a picture of the landscape of some of the important generalised parsing techniques (that can be applied to all context-free grammars) developed over the last 40 years.

Chapter 3

The development of generalised parsing

The field of parsing has been the focus of research for over 40 years, but we still have not found the holy grail of the parsing world – a linear time general parsing algorithm. We do not even know if it is possible to parse all context-free languages in linear time. This chapter is a tour of the major developments of generalised parsing and a discussion of the links between the different algorithms.

3.1 Overview

There are many different parsing algorithms described in the literature. By tracing their development, valuable insights can be gained. Techniques that may at first appear to be different, can often be related. For example, the CYK algorithm [CS70, You67, KT69] is the unification of several independently developed algorithms. Furthermore, the CYK algorithm has been shown [GH76] to be equivalent to the algorithm developed by Earley [Ear68]. This opinion has been voiced before by Dick Grune [GJ90],

> When we consult the extensive literature on parsing techniques, we seem to find dozens of them, yet there are only two techniques to do parsing; all the rest is technical detail and embellishment.

Many algorithms have been developed in response to limitations of, or to incorporate optimisations of, existing techniques. For example, GLR algorithms, initially described by Tomita [Tom86], are extensions of the standard LR algorithm. As we shall discuss in later chapters, Nozohoor-Farshi [NF91] removes the limitations of Tomita's algorithm, whilst Aycock and Horspool [AH99] incorporate Tomita's efficient data-structure to optimise a separate algorithm.

Recently, approaches that were previously deemed too inefficient are becoming practical solely due to the rapid increase of computing power. Tomita's algorithm was developed in the context of natural language parsing where input strings are typically short. At that time, the use of his algorithm for parsing programming languages was considered infeasible. However, as we shall see in Chapter 9, several commonly used tools implement variants of Tomita's algorithm.

This chapter presents an overview of the relationships between different generalised parsing techniques and provides two of the most straight-forward algorithms – Unger's algorithm and the CYK algorithm.

The diagram in Figure 3.1 lists several algorithms by the names of its developers, grouping similar approaches together in boxes. Solid arrows between algorithms indicate an extension or improvement made, while dotted arrows lead to the implementation of a specific algorithm.

The box on the top left of Figure 3.1 shows two early general context-free parsing algorithms. The technique described by Irons [Iro61] has been credited as being the "first fully described parser" [GJ90]. It is a full backtracking recursive-descent, left-corner parser that displays exponential worst case time complexity.

The second approach, attributed to Unger [Ung68], is a straightforward general parsing algorithm that has been "anonymously used" [GJ90] by many other algorithms. Although other algorithms, like CYK, are more efficient and have had more attention, Unger's algorithm has been modified by Sheil to achieve the same performance [She76]. It has the advantage of being extremely easy to understand and as such we give an overview of the technique in this chapter.

The existing (general) parsing algorithms were too inefficient to be used as programming language parsers. As a consequence of this inefficiency, two approaches were developed that considered only the deterministic subclass of the context-free grammars.

The LL grammars can be used to describe the syntax of a useful class of deterministic context-free grammars which include the syntax of many programming languages. Lewis and Stearns [LS68] are considered to be major contributors to the development of the top-down LL parsing technique. Although LL parsers are efficient (linear time) they do not accept left recursive grammars. It is often useful to define programming language grammars using left recursion and whilst standard removal algorithms exist [AU73], the structure, and potentially the semantics, of the grammar are altered by the transformation. However, the technique is still popular because it is fairly straightforward to generate LL parsers by hand.

At about the same time, Knuth developed his LR parsing algorithm [Knu65], a bottom-up approach (described in Chapter 2) which achieves linear time parsing of a context-free subclass larger than the class of LL grammars. In particular, they can cope with deterministic left recursive grammars. However, the required

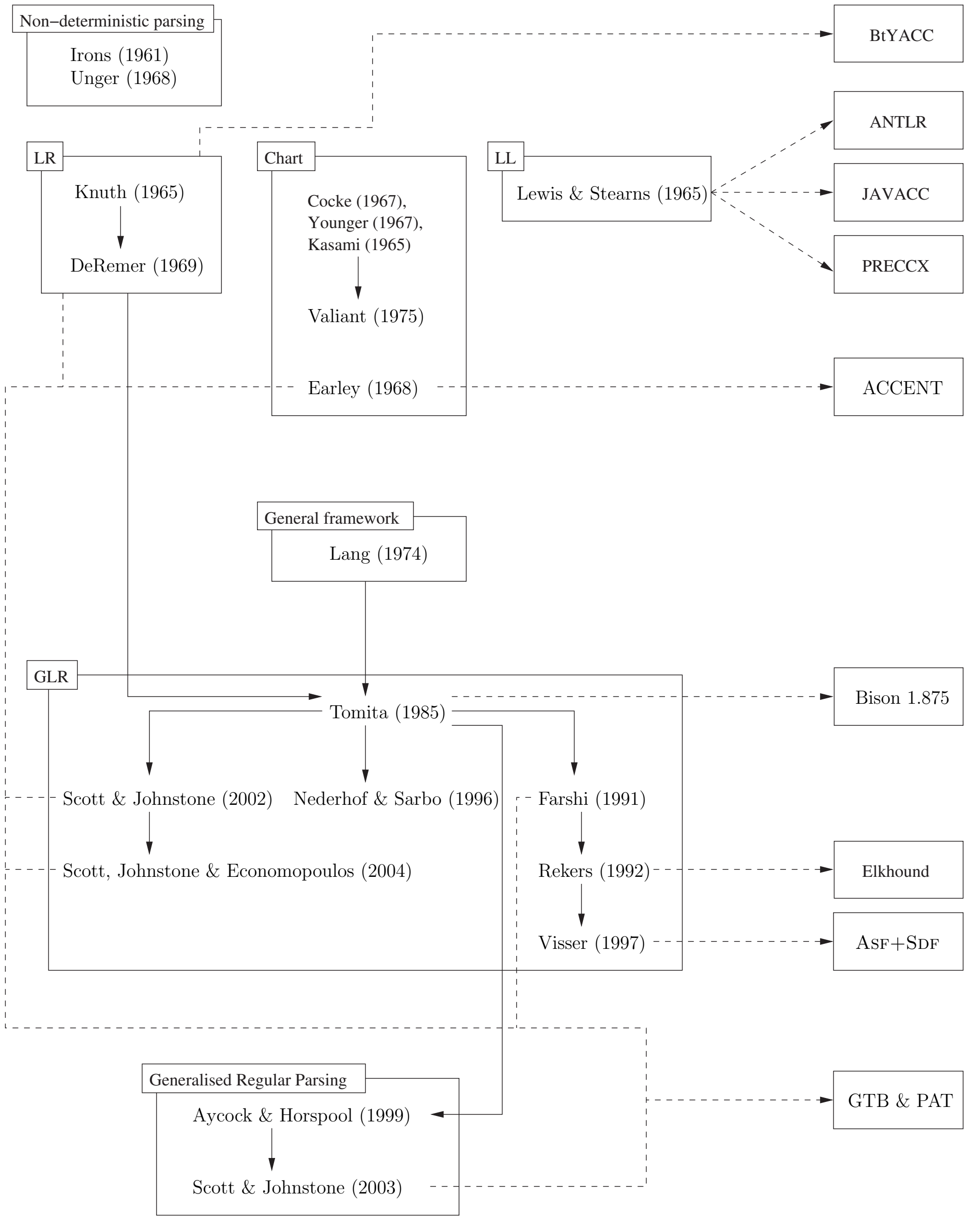


Figure 3.1: The development of general context-free parsing. Algorithms are grouped in boxes by contributing authors. Solid arrows indicate an extension or improvement made, while dotted arrows lead to the implementation of a specific algorithm.

automata were too large to be of practical use. Although smaller automata were developed [DeR69, DeR71], generating them by hand was a difficult and laborious task. The development of parser generators, in particular Yacc [Joh79], made LR parsing a popular approach for programming language parsers.

Whilst the programming language community focused on improving the efficiency of a useful (restricted) class of context-free grammars, the natural language processing community required more efficient parsers for all context-free grammars. The CYK algorithm was independently developed by Cocke [CS70], Younger [You67] and Kasami [Kas65] in the 1960's. The algorithm parses all context-free grammars in worst-case cubic time, but relies on grammars being in two-form. We discuss the CYK algorithm in more detail in Section 3.3 of this chapter.

Another technique developed later by Earley [Ear68] performs better in some cases but has a similar worst case complexity to CYK. Earley's approach is a directional, bottom-up technique. In this respect it appears to be very different to CYK, but it has been shown [GHR80] that the two algorithms are in fact closely related. We discuss Earley's algorithm in more detail in Chapter 8.

Many programming language developers were content with efficiently parsing a subset of the context-free grammars and new programming languages were designed with grammars that were 'easy' to parse. It was not until the 1980's that an interest in generalised parsing resurfaced in the form of Tomita's GLR parser [Tom86]. Tomita's algorithm extends the standard LR parsing technique to parse a larger class of grammars. Although his algorithm fails to terminate for certain grammars, it parses all LR grammars in linear time. Corrected versions of Tomita's algorithm have been given by Farshi [NF91], Scott & Johnstone [SJ00] and Nederhof & Sarbo [NS96]. We discuss these approaches in Chapters 4, 5 and 8 respectively.

There have been many attempts to speed up the performance of GLR parsers. A novel technique presented by Aycock and Horspool improves the efficiency through the reduction of stack activity [AH99]. Unfortunately their technique fails to work for certain grammars. However, an extension of their approach has been given by Scott and Johnstone [SJ03b] which successfully parses all context-free grammars. We discuss these algorithms in Chapter 7.

In the remainder of this chapter we shall outline two important developments in the history of parsing. They have no immediate impact on the GLR-style algorithms that are the main topic of this thesis and we shall not need them later. However, we include them for completeness.

3.2 Unger's method

One of the earliest and most straightforward general parsing algorithms is the technique developed by Unger [Ung68]. It has the advantage of being extremely easy to understand and as such we present an example parse in detail.

Example – recognition using Unger's method

Unger's method works by trying to partition the input string so that it can be derived from the start rule. For example consider Grammar 3.1.

$$\begin{aligned} &S' ::= S \\ &S ::= ASB \mid BSA \mid c \\ &A ::= a \\ &B ::= b \end{aligned} \tag{3.1}$$

We start off by partitioning the input string for the right hand side of the start symbol.

S
a b c a b

The non-terminal S can be replaced by one of 3 alternates. We start off with the first alternate and partition the input as follows.

S		
A	S	B
a	b	c a b
a	b c	a b
a	b c a	b
a b	c a	b
a b	c	a b
a b c	a	b

This method of partitioning can easily get out of hand if the right hand side of rules and the input string are large. Unger provided some optimisations that limited the number of partitions that need to be kept. Any partitions that do not match the terminals in the input string can be removed. This leaves us with only one partition that can possibly lead to a derivation of the input string. For example, since the non-terminals A and B cannot be extended any further, we focus on the non-terminal S, which has three alternates that can potentially be used to derive *bca*.

S		
A	S	B
a	b c a	b

Trying the first alternate we see that it cannot be used because the non-terminals A and B do not derive the terminal they have been partitioned with.

S		
A	S	B
b	c	a

Trying the next alternate, we have a success as all the non-terminals are able to derive the terminals in one step.

S		
B	S	A
b	c	a

This leads us to the following (unique) derivation of the input string.

S ⇒ ASB ⇒ aSB ⇒ aBSAB ⇒ abSAB ⇒ abcAB ⇒ abcaB ⇒ abcab

A naïve implementation of Unger's algorithm has exponential time complexity, which limits its use to trivial examples. The addition of a well-formed substring table dramatically improves the efficiency, the complexity becomes $O(n^{k+1})$ where n is the length of the input string and k is the maximum length of a rule's right hand side (see [GJ90] for more details).

3.3 The CYK algorithm

The CYK algorithm is a general recognition algorithm with $O(n^3)$ worst case time complexity for all context-free grammars in Chomsky Normal Form (CNF). A context-free grammar is said to be in CNF if every rule is of the form $A ::= BC$, or $A ::= a$, or $S ::= \epsilon$, where ABC are non-terminals and S is the grammar's start symbol.

The CYK algorithm uses an $(n + 1)(n + 1)$ triangular matrix, where n is the length of the input string, to determine whether a string is in the language. Each cell contains a set of non-terminals that are used in a derivation. In [GHR80] the matrix is constructed in the top right diagonal instead of the top left as is done by Cocke, Younger and Kasami. Both algorithms are equivalent, but Graham's description is easier to compare against other chart parsers, like Earley's algorithm. The formal specification of the recognition matrix construction algorithm, taken from [GHR80], is as follows.

CYK recognition matrix construction algorithm

```
**for** $i := 0$ **to** $n - 1$ **do**
    $t_{i,i+1} := \{A \mid A \to a_{i+1} \in P\}$
**for** $d := 2$ **to** $n$ **do**
    **for** $i := 0$ **to** $n - d$ **do**
        $j := d + i$
```

$t_{i,j} := \{A \mid$ there exists $k,\ i+1 \le k \le j-1$ such that $A \to BC \in P$ for some $B \in t_{i,k},\ C \in t_{k,j}\}$

Example – recognition using the CYK algorithm

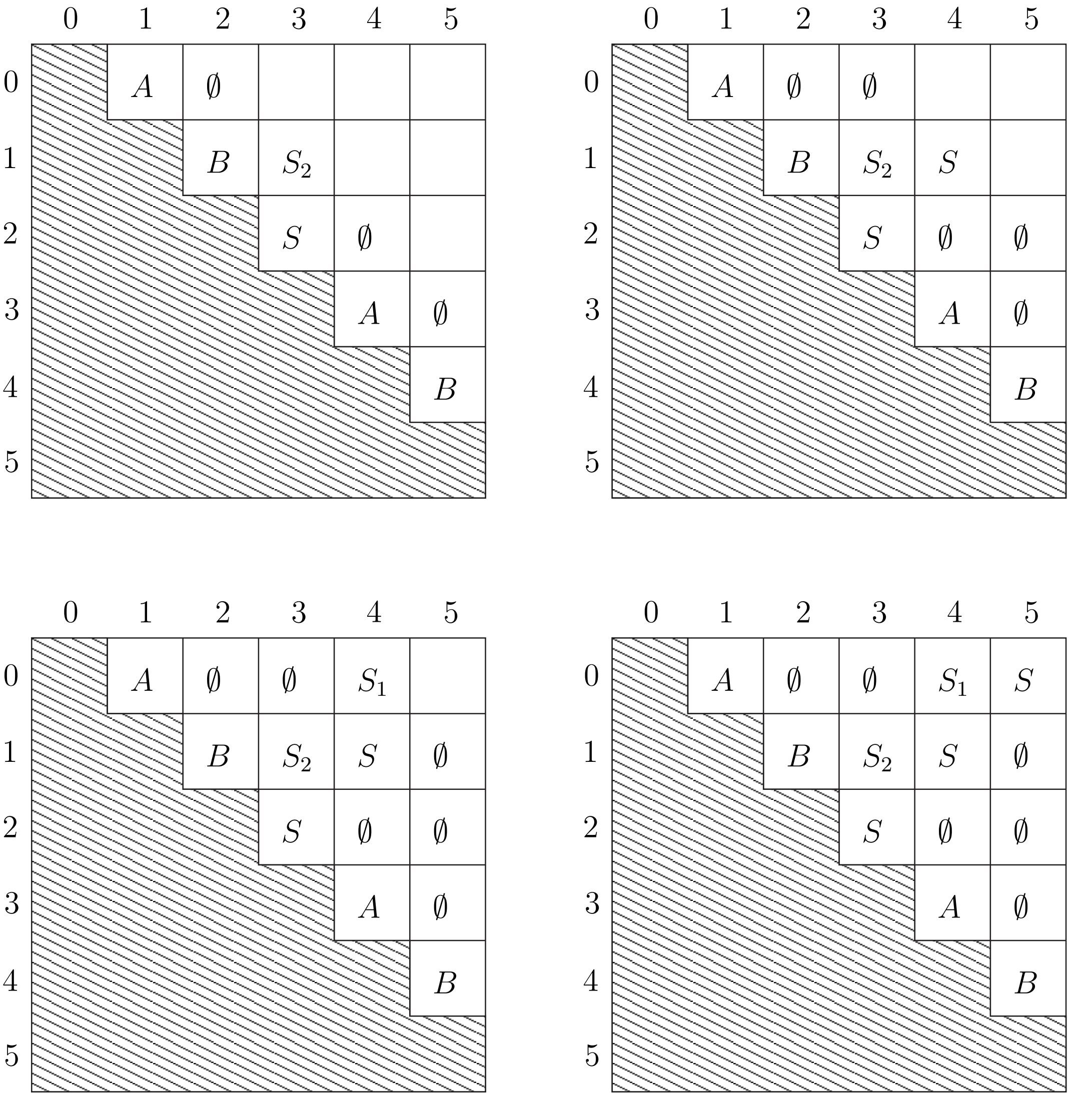
To demonstrate the recognition of a string using the CYK algorithm we trace the construction process of the recognition matrix for the string *abcab* in Grammar 3.1. Since the CYK algorithm requires grammars to be in CNF we use the CNF conversion algorithm presented in [AU73] to transform Grammar 3.1 to Grammar 3.2.

$$\begin{array}{ll} S' & ::= S \\ S & ::= S_1B \mid S_2A \mid c \\ S_1 & ::= AS \\ S_2 & ::= BS \\ A & ::= a \\ B & ::= b \end{array} \tag{3.2}$$

We begin the parse of the string *abcab* by filling the *superdiagonal stripe* [GH76] of the matrix from the top left to the bottom right of the matrix with non-terminals that directly derive the consecutive symbols of the input string.

	0	1	2	3	4	5
0		A				
1			B			
2				S		
3					A	
4						B
5						

The construction is continued by filling the entries of each subsequent diagonal in turn. The entry at (i, j) is filled with the set of non-terminals that define any rule whose right hand side is equal to the entries in (i, k) and (k, j) where $i+1 \le k \le j-1$. The completion of the CYK recognition matrix for our example parse is shown in Figure 3.2.



	0	1	2	3	4	5
0		A	$\emptyset$			
1			B	S_2		
2				S	$\emptyset$	
3					A	$\emptyset$
4						B
5						

	0	1	2	3	4	5
0		A	$\emptyset$	$\emptyset$		
1			B	S_2	S	
2				S	$\emptyset$	$\emptyset$
3					A	$\emptyset$
4						B
5						

	0	1	2	3	4	5
0		A	$\emptyset$	$\emptyset$	S_1	
1			B	S_2	S	$\emptyset$
2				S	$\emptyset$	$\emptyset$
3					A	$\emptyset$
4						B
5						

	0	1	2	3	4	5
0		A	$\emptyset$	$\emptyset$	S_1	S
1			B	S_2	S	$\emptyset$
2				S	$\emptyset$	$\emptyset$
3					A	$\emptyset$
4						B
5						

Figure 3.2: Trace of construction of CYK recognition matrix for parse of string *abcab* in Grammar 3.2.

Once the construction of the recognition matrix is complete, we check the entry in the top right cell. If it contains the non-terminal on the right hand side of the start rule then the string is accepted. In our example parse we can see that the string *abcab* is in the language of Grammar 3.2.

3.4 Summary

In this chapter we have discussed some of the major developments of generalised parsing techniques. We also briefly discussed Unger's approach and the CYK recognition algorithm.

In the next chapter we discuss, in detail, Tomita's GLR parsing algorithm and the extensions due to Farshi and Rekers.

Chapter 4

Generalised LR parsing

As we have already discussed in earlier chapters, context-free grammars were developed by Noam Chomsky in the 1950's in an attempt to capture the key properties of human languages. Computer scientists took advantage of their declarative structure and used them to define the syntax of programming languages. Many parsing algorithms capable of parsing all context-free grammars were developed, but their poor worst case performance often made then impractical. As a result more efficient parsing algorithms which worked on a subclass of the context-free grammars were developed.

The efficiency of the contemporary generalised parsing techniques (CYK, Earley, etc.) were disappointing when compared to Knuth's deterministic LR parsing algorithm. Its popularity had soared, but the class of grammars it accepted was too restrictive for the practical applications that interested Tomita − natural language processing. In 1985, he developed the GLR parsing algorithm by extending the LR algorithm to work on non-LR grammars.

This chapter presents Tomita's GLR recognition and parsing algorithms [Tom86, Tom91]. Although these algorithms work for a larger class of grammars than the LR parsers they cannot correctly parse all context-free grammars. We discuss the modification due to Farshi [NF91], that extends Tomita's recognition algorithm to work for all context-free grammars and then discuss Rekers' [Rek92] parser extension.

4.1 Using the LR algorithm to parse all context-free grammars

In Chapter 2 we described the standard LR parsing algorithm. LR parsers are extremely efficient, but are restricted by the class of grammars they accept. Although they work for a useful subset of the context-free grammars, they cannot cope with non-determinism.

A naïve approach to dealing with non-determinism in an LR parser is to duplicate a stack when a conflict in the parse table is encountered. An approach presented in [Tom84] uses a *stack list* to represent the different stacks of a non-deterministic parse. Each stack in the stack list is controlled by a separate LR parsing process that works in the same way as the standard LR algorithm. However, each process essentially works in parallel by synchronising on the shift actions.

Each stack in the stack list is represented by a graph, where the nodes are labelled with a state number and the edges are labelled with the symbol parsed. The rightmost nodes are the tops of each stack. For example, consider the parse of the string *abd* for the ambiguous Grammar 4.1 and the DFA in Figure 4.1. The associated LR(1) parse table, with conflicts, is shown in Table 4.1.

$$\begin{aligned}&0.\ S' ::= S\\&1.\ S ::= abC\\&2.\ S ::= aBC\\&3.\ B ::= b\\&4.\ C ::= d\end{aligned} \tag{4.1}$$

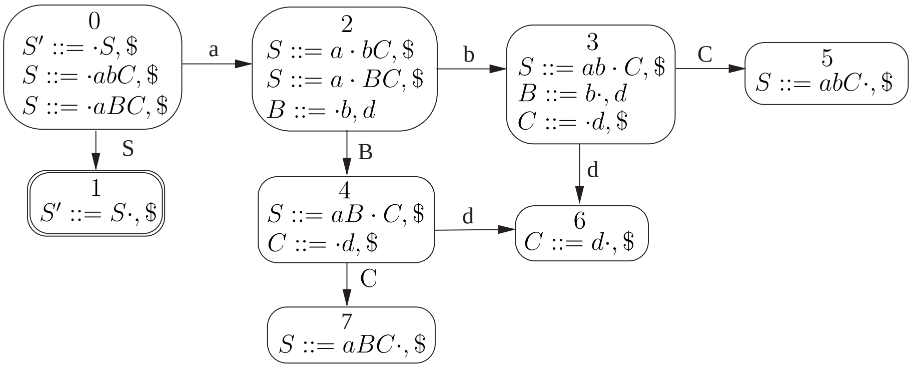
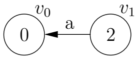


Figure 4.1: The DFA for Grammar 4.1.

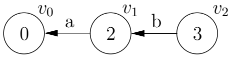
	a	b	d	$	B	C	S
0	s2					g1	
1				acc			
2		s3			g4		
3			s6/r3			g5	
4			s6			g7	
5				r1			
6				r4			
7				r2			

Table 4.1: The parse table for Grammar 4.1.

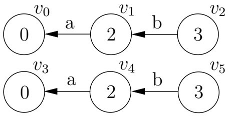
We begin the parse by creating the start node, v_0, of the stack list, labelled by the start state of the DFA. We read the first input symbol, a, and then perform the shift from state 0 to state 2. We create a new node, v_1 in the stack list, labelled 2, and add an edge, labelled a back to v_0.



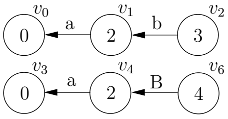
We then proceed to read the next input symbol, b, and create the node, v_2, labelled 3 with an edge back to v_1.



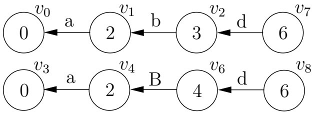
At this point we are in state 3 of the DFA which contains a shift/reduce conflict. Since we do not know which action to perform, we duplicate the stack and perform both actions.



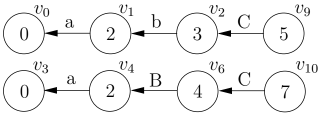
We synchronise the stack list on shift actions, so we perform the reduce $B ::= b\cdot$ on the bottom stack first. This involves popping the node v_5 off the stack and adding a new node v_6, labelled 4, with an edge labelled B back to v_4.



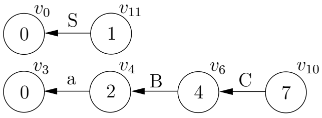
We then read the next input symbol, d, and perform the synchronised shift action, $s6$, for both stacks. We create the new nodes v_7 and v_8, with edges labelled d, back to v_2 and v_6 respectively.



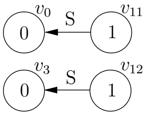
Since there is a reduce on rule $C ::= d$ from both states at the top of the stack list we can perform both to get the new stack list shown below.



From state 5 there is a reduce on $S ::= abC$ so we pop the top 3 nodes off the first stack and create the new node, v_{11}, labelled 1, with an edge from v_{11} to v_0.



From state 7 there is a reduction on rule $S ::= aBC$, so we pop the top 3 nodes off the second stack as well and create the new node, v_{12} as shown below.

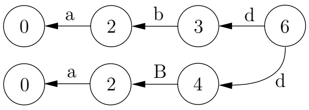


Since all the input has been consumed and v_{10} and v_{11}, which are labelled by the accept state of the DFA, are at the top of the stack list the parse has succeeded.

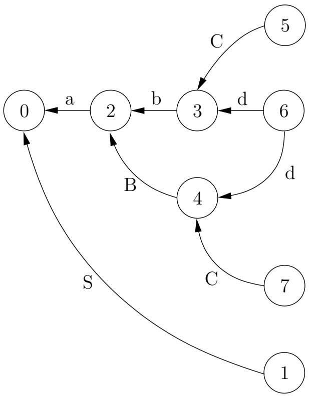
Using a stack list to follow all parses of an ambiguous sentence can be very inefficient. If a state can be reached in several different ways, because of non-determinism, then there will be several duplicate stacks. Unfortunately, this can cause the number of stacks created to grow exponentially. In addition to this, because the stacks do not share any information with each other, if the tops of several stacks contain the same state then they continue to parse the remaining input in the same way until the duplicate states are popped off each stack.

Such redundant actions can be prevented with the use of a slightly modified structure called a Tree Structured Stack (TSS) [Tom84]. When the tops of two or more stacks contain the same state, a single state is shared between each stack. This prevents duplicate parses of the same input being done.

For example, consider the parse of the string *abd* shown above. After duplicating the stack and performing the first reduction the next action is a shift on d. Since both stacks reach state 6 on the shift we can merge the nodes v_8 and v_9 to combine the stack list into a TSS.



Although this approach significantly improves the efficiency of the algorithm, it is still far from ideal – the number of stacks created can still grow exponentially. In both approaches discussed above, when a conflict in the parse table is encountered the entire stack is duplicated. However, it is often the case that separate stacks have the same nodes towards the bottom of the stack (the leaves of the TSS). Instead of duplicating a stack when a non-deterministic point in the parse is reached, the space required can be reduced by only splitting the necessary part of the stack. The resulting structure is called a Graph Structured Stack (GSS) [Tom86]. The GSS constructed for the parse of the string *abd* is shown below.



The next section presents Tomita's Generalised LR (GLR) algorithm that extends the standard LR parser by constructing a GSS during a parse.

4.2 Tomita's Generalised LR parsing algorithm

In [Tom86], Tomita presents five separate GLR algorithms which we shall refer to as Algorithms 0–4. The first four algorithms are recognisers and the fifth algorithm is the extension of Algorithm 3 to a parser. Algorithm 0 is defined to work for LR(1) grammars and is used to introduce the construction of the GSS. Algorithm 1 extends Algorithm 0 to work for non-LR(1) grammars without ϵ-rules. Algorithm 2 introduces a complex approach to deal with ϵ-rules and forms the basis of Algorithm 3 – the full version of the recognition algorithm. Algorithm 4 is the parser version of the GLR algorithm which constructs a shared packed parse forest representation of parse trees.

This section introduces Tomita's recognition Algorithms 1 and 2. We begin by defining the central data structure that underpins all of Tomita's algorithms – the GSS. We demonstrate the construction of the GSS for both algorithms and highlight some of the problems associated with each of the approaches. Algorithms 3 and 4 are discussed in Section 4.4.

4.2.1 Graph structured stacks

A Graph Structured Stack (GSS) is the central data structure that underpins all of Tomita's GLR algorithms. An instance of a GSS is related to a specific grammar Γ and input string $a_1 \dots a_n$. It is defined as a directed acyclic graph, containing two types of node: state nodes, labelled by the state numbers of the DFA for Γ and symbol nodes, labelled by Γ's grammar symbols.

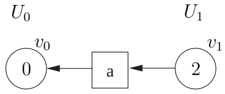
The state nodes are grouped together into $n + 1$ disjoint sets called *levels.* The GSS is constructed one level at a time. First all possible reductions are performed for the state nodes in the current level (the frontier), and then the next level is created as a result of applying shift actions. The first level is initialised with a state node labelled by the DFA's start state.

We represent a GSS graphically by drawing the state nodes as circular nodes and the symbol nodes are square nodes. To separate each of the levels we draw the state nodes of a given level in a single column labelled U_i, where $0 \leq i \leq n$. The GSS is drawn from left to right, with the rightmost nodes representing the tops of each of the stacks.

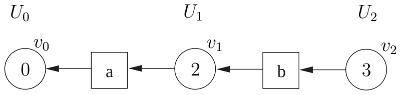
4.2.2 Example – constructing a GSS

We shall describe how a GSS is constructed during the parse of a string *abc* with Grammar 4.1, whose DFA shown in Figure 4.1 and the associated parse table in Table 4.1. We shall refer to the parse table as $\mathcal{T}$.

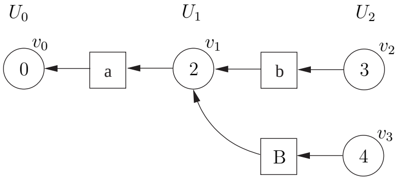
The GSS is initialised with the state node, v_0, in level U_0. For each new state node constructed in the GSS we check to see what actions are applicable. We begin the parse by finding the shift action s2 in $\mathcal{T}(0, a)$. This triggers the creation of the next level, U_1 for which we create the new node, v_1, labelled 2. We create a new symbol node, labelled a, and make it the successor of v_1 and the predecessor of v_0.



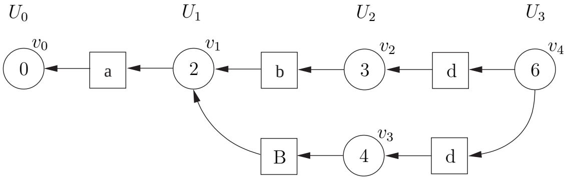
We process v_1 and find the shift action s3 in $\mathcal{T}(2, b)$. We construct the new node, v_2 labelled 3 and add it to level U_2. We read the b from the input string, construct the new symbol node labelled b and make it the successor of v_2 and the predecessor of v_1.



Next we process v_3. In $\mathcal{T}(3, c)$ there is a shift/reduce conflict, s6/r3. Since the construction of the GSS is synchronised on the shift actions, we queue the shift, and continue by performing the reduction by rule 3, $B ::= b$. Unlike the standard LR parser that removes states from the parse stack when a reduction is performed, a GLR algorithm does not remove nodes from the GSS[1]. Instead of popping nodes off the stack, we perform a traversal of all *reduction paths* of length 2, from node v_2, to find the target nodes of the reduction. In this case there is only one path, which leads to node v_1. We find the goto action, g4 $\in \mathcal{T}(2, B)$, and create the new state node, v_3, labelled 4 in the current level U_2. We then create the symbol node labelled B and make it the successor of v_3 and the predecessor of v_1.

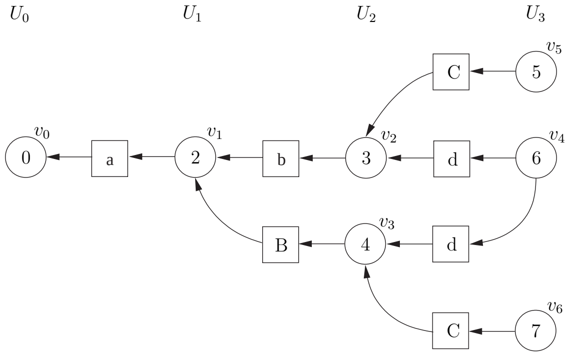


The only action associated with the newly created node, v_3, is the shift s6. Since we have processed all nodes in U_2 and performed all possible reductions the current level is complete. So next we construct level U_3 by performing the shift actions from v_2 and v_3. We create the new node, v_4, labelled 6, in U_3 and two new symbol nodes labelled d. We then add two paths of length 2 from v_4, one going to v_2 and the other going to v_3.

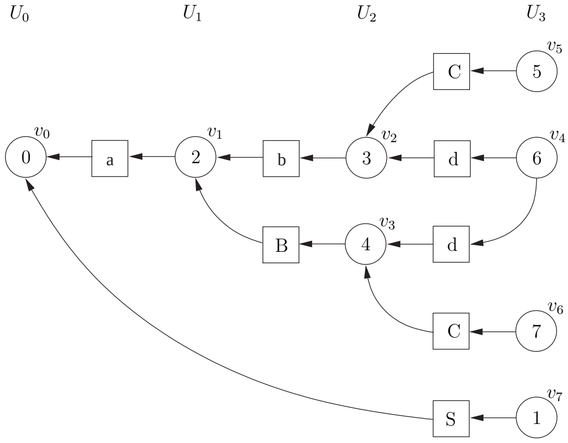


At this point we have consumed all the input symbols and are left with the lookahead symbol \$. We process v_4 and find the reduce action r4 $\in \mathcal{T}(6, \$)$. Since the right hand side of rule 4 contains one symbol we trace back paths of length 2 from v_4. In this case two possible paths exist; one reaching v_2 and the other v_3. We create two new nodes, v_5 and v_6, labelled with the goto states found in $\mathcal{T}(3, C)$ and $\mathcal{T}(4, C)$ respectively. We create two new symbol nodes, both labelled C and use them to create a path between v_5 and v_2 and v_6 and v_3.

[1] Although it is possible to perform garbage collection to remove 'dead' nodes (nodes that can no longer be reached on a path through the GSS from the current level) we do not address the issue here. For a technique that implements such an approach see [MN04].



Processing both nodes v_5 and v_6 we see that the reduction r1 is applicable. The right hand side of rule 1 consists of 3 symbols, so we need to find the nodes at the end of the paths of length 6. Both reduction paths reach v_0 and since the associated goto action is the same for both reductions we create one new node, v_7, labelled 1 in the current level. Only one symbol node labelled S is required, with a path from v_7 to v_0.



Processing node v_7 we find the accept action $acc \in \mathcal{T}(1, \$)$. Since we have consumed all of the input and processed all state nodes in the current level the input string *abc* is accepted.

Note that it is a property of the GSS that all symbol nodes pointed to by a state are labelled by the same grammar symbol.

4.2.3 Tomita's Algorithm 1

Tomita's Algorithm 1 basically works by constructing a GSS in the way we have described in the previous section. However, as is shown in the previous example there may be more than one node in the frontier of the GSS waiting to be processed. Tomita uses a special bookkeeping set, $\mathcal{A}$, to keep track of these newly created state nodes. A parse is performed by iterating over this set and finding all applicable actions for a given node. However, as it is possible for a node to have multiple applicable actions (as a result of a conflict in the parse table) two additional bookkeeping sets, $\mathcal{Q}$ and $\mathcal{R}$, are used to store any pending shifts and reductions. The set $\mathcal{Q}$ stores elements of the form (v, k), where v is the node labelled h that has a transition labelled by the next input symbol to a state k in the DFA.

Before describing the elements stored in the set $\mathcal{R}$ we discuss how reductions are performed in the GSS. Recall that the standard LR parser performs a reduction for a rule $A ::= X_1 \ldots X_j$ by popping j symbols off the top of the stack. In comparison a reduction in a GLR parser is associated with a node in the frontier and requires all paths of length $2j$ to be traced back from the given node. In the worst case this search may require $O(n^j)$ time.

The efficiency of Tomita's algorithms stems from the fact that the same part of the input is not parsed more than once in the same way. In other words, each reduction path is only traversed once for each reduction. However, it is possible for a new edge to be added to an existing node in the frontier that has already performed its associated reductions. This new edge introduces a new reduction path that needs to be traversed for the parse to be correct. So as not to traverse the same path more than once Tomita stores elements of the form (w, t), where t is the rule number of the reduction in the set $\mathcal{R}$ and w is the first edge of the path down which reduction t is to be applied.

Below is the formal description of Tomita's Algorithm 1, taken from [Tom86].

Algorithm 1

```
PARSE(G, $a_1 \ldots a_n$)
  • $\Gamma \Leftarrow \emptyset$
  • $a_{n+1} \Leftarrow$ '\$'
  • $r \Leftarrow$ FALSE
  • create in $\Gamma$ a vertex $v_0$ labelled $s_0$.
  • $U_0 \Leftarrow \{v_0\}$
  • for $i \Leftarrow 0$ to n do PARSEWORD($i$).
  • return $r$.

PARSEWORD($i$)
  • $\mathcal{A} \Leftarrow U_i$.
  • $\mathcal{R}, \mathcal{Q} \Leftarrow \emptyset$.
  • repeat
      ◦ if $\mathcal{A} \neq \emptyset$ then do ACTOR.
      ◦ elseif $\mathcal{R} \neq \emptyset$ then do REDUCER.
  • until $\mathcal{R} = \emptyset \wedge \mathcal{A} = \emptyset$.
• do SHIFTER.

ACTOR
  • remove one element $v$ from $\mathcal{A}$.
  • for all $\alpha \in$ ACTION(STATE($v$), $a_{i+1}$) do
    ○ if $\alpha$ = 'accept' then $r \Leftarrow$ TRUE.
    ○ if $\alpha$ = 'shift s' then add $\langle v, s \rangle$ to $\mathcal{Q}$.
    ○ if $\alpha$ = 'reduce p' then
      • for all $x$ such that $x \in$ SUCCESSORS($v$), add $\langle v, x, p \rangle$ to $\mathcal{R}$.

REDUCER
  • remove one element $\langle v, x, p \rangle$ from $\mathcal{R}$.
  • N $\Leftarrow$ LEFT($p$)
  • for all $w$ such that there exists a path of length $2 * |p| - 1$ from $x$ to $w$ do
    ○ $s \Leftarrow$ GOTO(STATE($w$),N)
    ○ if there exists $u$ such that $u \in U_i \wedge$ STATE($u$) $= s$ then
      • if there already exists a path of length 2 from $u$ to $w$ then
        ○ do nothing.
      • else
        ○ create in $\Gamma$ a vertex $z$ labelled N.
        ○ create two edges in $\Gamma$ from $u$ to $z$ and from $z$ to $w$.
        ○ if $u \notin \mathcal{A}$ then
          • for all $q$ such that 'reduce $q$' $\in$ ACTION(STATE($u$),$a_{i+1}$) do add
            $\langle u, z, q \rangle$ to $\mathcal{R}$.
        ○ else /* if there doesn't exist $u$ such that $u \in U_i \wedge$ STATE($u$) $= s$ */
          • create in $\Gamma$ two vertices $u$ and $z$ labelled $s$ and N, respectively.
          • create two edges in $\Gamma$ from $u$ to $z$ and from $z$ to $w$.
          • add $u$ to both $\mathcal{A}$ and $U_i$.

SHIFTER
  • for all $s$ such that $\exists v(\langle v, s \rangle \in \mathcal{Q})$,
    ○ create in $\Gamma$ a vertex $w$ labelled $s$.
    ○ add $w$ to $U_{i+1}$.
    ○ for all $v$ such that $\langle v, s \rangle \in \mathcal{Q}$ do
      • create $x$ labelled $a_{i+1}$ in $\Gamma$.
      • create edges in $\Gamma$ from $w$ to $x$ and from $x$ to $v$.
```

4.2.4 Parsing with ϵ-rules

Tomita's Algorithm 1 is defined to work on ϵ-free grammars and hence does not contain the machinery to deal with ϵ-rules. However, by modifying the way reductions are performed, the algorithm can parse grammars containing ϵ-rules. Recall that when a new node, v, is created whose state contains an applicable reduction for a rule of the form $A ::= \alpha$, the algorithm finds all nodes at the end of the paths of length $2 \times |\alpha| - 1$ from the successors of v. Since ϵ-rules have length zero, an ϵ-reduction does not require a path to be traversed.

Although it is trivial to extend Algorithm 1 to deal with ϵ-rules, the straightforward approach fails to parse certain grammars correctly. For example, consider Grammar 4.2 and the string aab. The associated LR(1) DFA is shown in Figure 4.2.

$$\begin{aligned} S' &::= S \\ S &::= aSB \mid b \\ B &::= \epsilon \end{aligned} \tag{4.2}$$

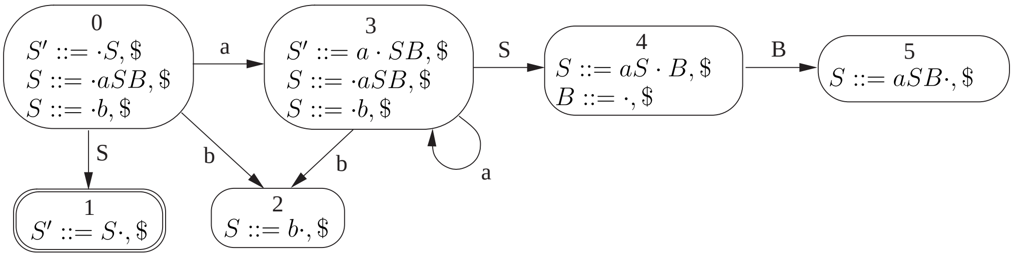
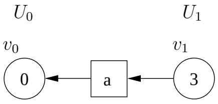
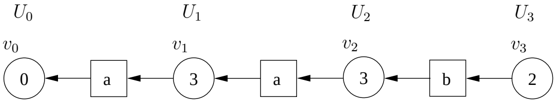


Figure 4.2: The LR(1) DFA for Grammar 4.2.

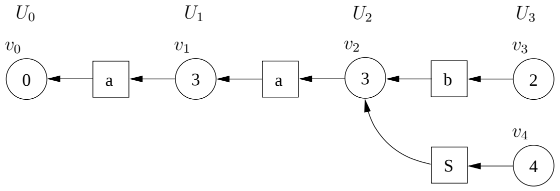
We begin the parse by creating v_0 and adding it to U_0 and the set $\mathcal{A}$. When v_0 is processed in the ACTOR we only find a shift to state 3 on the first input symbol a. We create a new symbol node labelled a which we make a successor of v_1 and a predecessor of v_0.



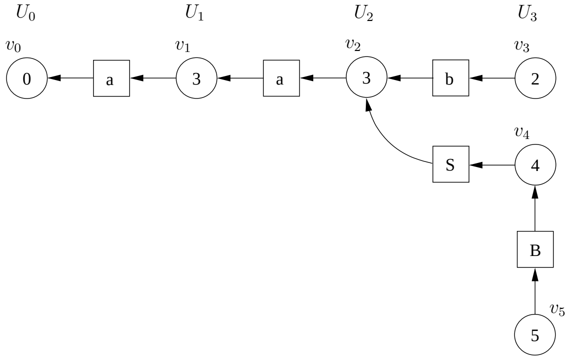
We continue in this way shifting the next two input symbols and constructing the GSS shown below.



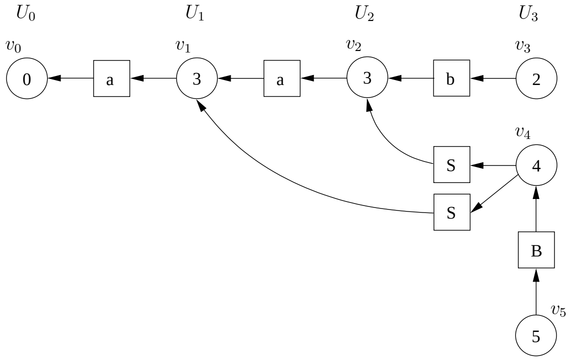
Processing v_3, we find the reduction on rule 2, $S ::= b$, which is applicable from state 2. From the only successor of v_3, the symbol node labelled b, we traverse a path of length one to v_2. We then create the new state node v_4, labelled 4, and a new path of length two between v_4 and v_2 via a new symbol node labelled S.



From state 4 there is only one applicable reduction on rule 3, $B ::= \epsilon$. Since the right hand side of the rule is ϵ, we do not traverse a reduction path. This results in the creation of v_5 with a path to v_4 via a the new symbol node labelled B.



Processing v_5, we find a reduction on rule 1, $S ::= aSB$. We trace back a path of length 5 to v_1 from the successor of v_5. Since the node v_4, which is labelled by the goto state of the reduction, already exists in the frontier of the GSS a new path of length two is added from v_4 to v_1 via a new symbol node labelled S.



At this point there are no nodes waiting to be processed and no actions are queued in either of the bookkeeping sets $\mathcal{Q}$ or $\mathcal{R}$. Although all the input has been consumed the parse terminates in failure since there is not a state node in the frontier of the GSS that is labelled by the accept state of the DFA. This is clearly incorrect behaviour since the string aab is in the language of Grammar 4.2.

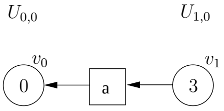
It turns out that the problem is caused by the new reduction paths created by nullable reductions. Recall that when a new edge is added from an existing node v

in the GSS, Algorithm 1 re-performs any reductions that have already been applied from v. However, in the above example the new path of length two from v_4 to v_1 did not create a new reduction path from v_4, but it did from v_5.

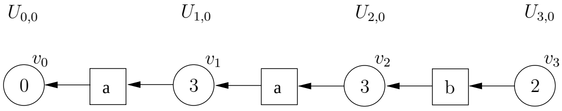
We discuss this problem in more detail in Chapter 5. Tomita deals with this problem in Algorithm 2 by introducing sub-levels to the GSS. When an ϵ-reduction is performed a new sub-frontier is created and the node labelled by the goto state of the reduction is created in this new sub-frontier. Before presenting the formal specification of Algorithm 2, we demonstrate its operation using the above example once again.

4.2.5 Tomita's Algorithm 2

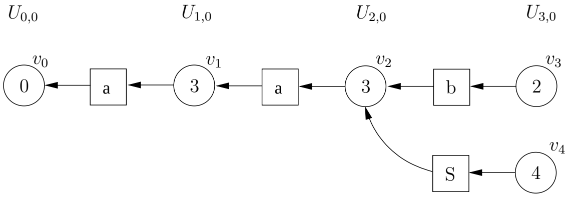
Tomita's Algorithm 2 creates sub-frontiers $U_{i,j}$ in U_i when ϵ-reductions are applied. To parse the string aab we begin by creating v_0, labelled with the start state of the DFA, in level $U_{0,0}$. The only applicable action from state 0 of the DFA is a shift to state 3 on the first input symbol, a. We read the a from the input string, create the symbol node labelled a and the state node v_1, labelled 3. We make the new symbol node a successor of v_1 and a predecessor of v_0. The GSS constructed up to this point is shown below.



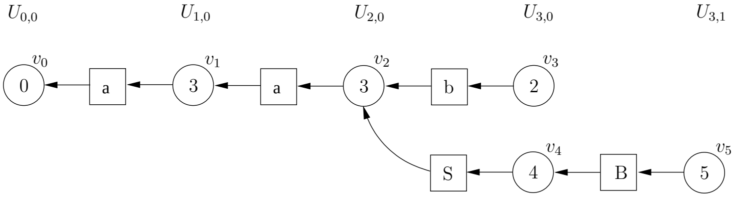
We continue in this way, reading the next two input symbols and constructing the GSS shown below.



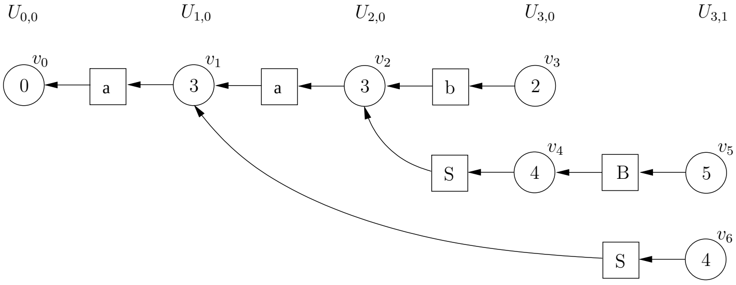
At this point state 2 contains a reduction on rule $S ::= b$. We perform the reduction by traversing a path of length two from v_3 to v_2. Since there is no node labelled by the goto state of the reduction we create v_4, the symbol node labelled S and the new path from v_4 to v_2.



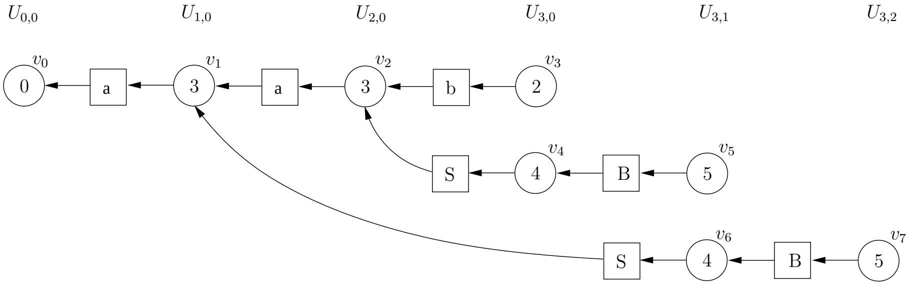
The only applicable action in state 4 is the nullable reduction on rule $B ::= \epsilon$. Since it is nullable reductions that cause new edges to be added to existing nodes, a new sub-frontier $U_{3,1}$ is created. We create the symbol node labelled B and the state node, v_5, in $U_{3,1}$. We make the new symbol node a successor of v_5 and a predecessor of v_4.



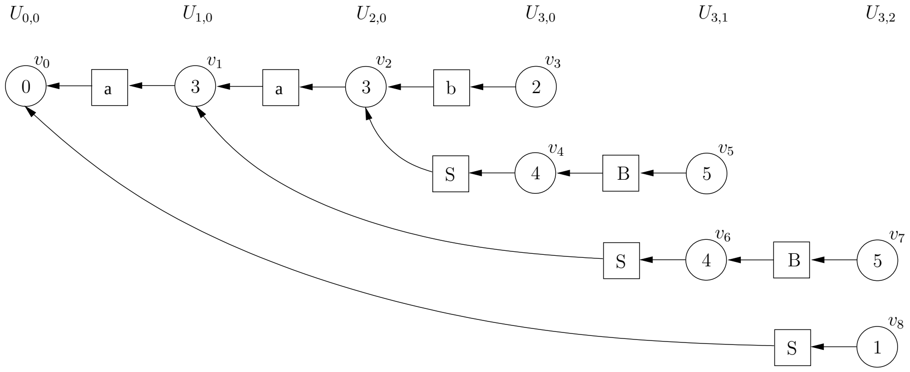
We then continue as normal processing any applicable actions from state 5. We trace back a path of length six for the reduction on rule $S ::= aSB$. Because the reduction is performed from a state node in the sub-frontier $U_{3,1}$ we also create the state node v_6, labelled by the goto state of the reduction in $U_{3,1}$.



From state 4 we can perform another nullable reduction on rule $B ::= \epsilon$. As a result we create a new sub-frontier, $U_{3,2}$, and add, v_7, the state node labelled by the goto state of the reduction to it.



Performing the reduction from v_7 on rule $S ::= aSB$, we trace back a path to v_0. We create, v_8, labelled 1, and the symbol node labelled S. We make the symbol node a successor of v_8 and a predecessor of v_0. Since v_8 is labelled by the accept state of the DFA and we have consumed all the input string, the parse is successful. The final GSS constructed is shown below.



Algorithm 2

```
PARSE(G, $a_1 \dots a_n$)
  • $\Gamma \Leftarrow \emptyset$
  • $a_{n+1} \Leftarrow$ '$'
  • $r \Leftarrow$ FALSE
  • create in $\Gamma$ a vertex $v_0$ labelled $s_0$.
  • $U_{0,0} \Leftarrow \{v_0\}$
  • for $i \Leftarrow 0$ to n do PARSEWORD($i$).
  • return $r$.

PARSEWORD($i$)
  • $j \Leftarrow 0$.
  • $\mathcal{A} \Leftarrow U_{i,0}$.
  • $\mathcal{R}, \mathcal{Q}, \mathcal{R}_e \Leftarrow \emptyset$.
  • repeat
      ◦ if $\mathcal{A} \neq \emptyset$ then do ACTOR
      ◦ elseif $\mathcal{R} \neq \emptyset$ then do REDUCER
      ◦ elseif $\mathcal{R}_e \neq \emptyset$ then do E-REDUCER
  • until $\mathcal{A} = \emptyset \wedge \mathcal{R} = \emptyset \wedge \mathcal{R}_e = \emptyset$.
  • do SHIFTER.

ACTOR
  • remove one element $v$ from $\mathcal{A}$.
  • for all $\alpha \in$ ACTION(STATE($v$), $a_{i+1}$) do
     ◦ if $\alpha$ = 'accept' then $r \Leftarrow$ TRUE.
     ◦ if $\alpha$ = 'shift $s$' then add $\langle v, s \rangle$ to $\mathcal{Q}$.
     ◦ if $\alpha$ = 'reduce $p$' and $p$ is not an e-production then
        • for all $x$ such that $x \in$ SUCCESSORS($v$), add $\langle v, x, p \rangle$ to $\mathcal{R}$.
     ◦ if $\alpha$ = 'reduce $p$' and $p$ is an e-production then add $\langle v, p \rangle$ to $\mathcal{R}_e$.
```

REDUCER

```
• remove one element $\langle v, x, p \rangle$ from $\mathcal{R}$.
• N $\Leftarrow$ LEFT($p$)
• **for all** $w$ such that there exists a path of length $2 * |p| - 1$ from $x$ to $w$ **do**
    ○ $s \Leftarrow$ GOTO(STATE($w$),N).
    ○ **if** there exists $u$ such that $u \in U_{i,j} \wedge$ STATE($u$) $= s$ **then**
        • **if** there already exists a path of length 2 from $u$ to $w$ **then**
            ○ do nothing.
        • **else**
            ○ create in $\Gamma$ a vertex $z$ labelled N.
            ○ create two edges in $\Gamma$ from $u$ to $z$ and from $z$ to $w$.
            ○ **if** $u \notin \mathcal{A}$ **then**
                • **for all** $q$ such that 'reduce $q$' $\in$ ACTION(STATE($u$),$a_{i+1}$) **do** add
                  $\langle u, z, q \rangle$ to $\mathcal{R}$.
            ○ **else** /* if there doesn't exist $u$ such that $u \in U_{i,j} \wedge$ STATE($u$) $= s$ */
                • create in $\Gamma$ two vertices $u$ and $z$ labelled $s$ and N, respectively.
                • create two edges in $\Gamma$ from $u$ to $z$ and from $z$ to $w$.
                • add $u$ to both $\mathcal{A}$ and $U_{i,j}$.
```

E-REDUCER

```
• $U_{i,j+1} \Leftarrow \emptyset$
• **for all** $\langle v, p \rangle \in \mathcal{R}_e$ **do**
    ○ N $\Leftarrow$ LEFT($p$).
    ○ $s \Leftarrow$ GOTO(STATE($w$),N).
    ○ **if** there exists $w$ such that $w \in U_{i,j+1} \wedge$ STATE($w$) $= s$ **then**
        • create in $\Gamma$ a vertex $x$ labelled N.
        • create edges in $\Gamma$ from $w$ to $x$ and from $x$ to $v$.
    ○ **else**
        • create in $\Gamma$ vertices $w$ and $x$ labelled $s$ and N.
        • create edges in $\Gamma$ from $w$ to $x$ and from $x$ to $v$.
        • add $w$ to $U_{i,j+1}$.
• $\mathcal{R}_e \Leftarrow \emptyset$.
• $\mathcal{A} \Leftarrow U_{i,j+1}$.
• $j \Leftarrow j + 1$.
```

SHIFTER

```
• $U_{i+1,0} \Leftarrow \emptyset$.
• **for all** $s$ such that $\exists v(\langle v, s \rangle \in \mathcal{Q})$,
    ○ create in $\Gamma$ a vertex $w$ labelled $s$.
    ○ add $w$ to $U_{i+1,0}$.
○ for all $v$ such that $\langle v, s\rangle \in \mathcal{Q}$ do
    • create $x$ labelled $a_{i+1}$ in $\Gamma$.
    • create edges in $\Gamma$ from $w$ to $x$ and from $x$ to $v$.
```

4.2.6 The non-termination of Algorithm 2

Although Algorithm 2 is defined to work on grammars containing ϵ-rules, it can fail to terminate when parsing strings with hidden-left recursive grammars. For example, consider the parse for the string ab with Grammar 4.3 and the LR(1) DFA in Figure 4.3.

$$\begin{aligned} S' &::= S \\ S &::= BSa \mid b \\ B &::= \epsilon \end{aligned} \tag{4.3}$$

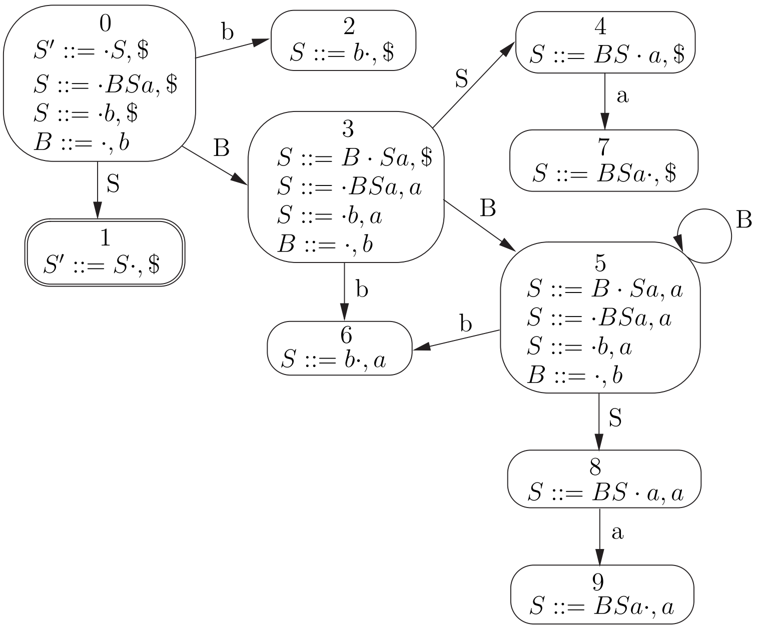
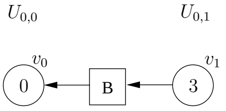
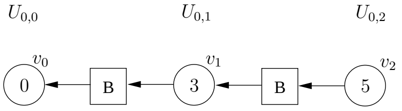


Figure 4.3: The LR(1) DFA for Grammar 4.3.

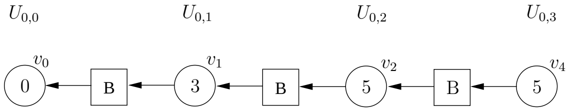
We begin the parse, as usual, by creating v_0 in level $U_{0,0}$. Since state 0 contains a shift/reduce conflict, we queue the shift and perform the reduction on rule $B ::= \epsilon$. Since the reduction is nullable we construct a new sub-frontier $U_{0,1}$ with the new state node, v_1, labelled 3 and a path of length two from v_1 to v_0, via the new intermediate symbol node labelled B.



State 3 contains another shift/reduce conflict, so we queue the shift and perform the reduction $B ::= \epsilon$ once again. Since this reduction is also nullable we create $U_{0,2}$, v_2 and a path of length two from v_2 to v_1 via the intermediate symbol node B.



Processing v_2, we find the same shift/reduce conflict in state 5; a shift on b and the nullable reduction on rule $B ::= \epsilon$. Performing this reduction results in yet another new sub-frontier and another state node labelled 5. This process continues indefinitely preventing the algorithm from terminating.



In the next section we discuss a modification of Algorithm 2 that enables the parser to work for all context-free grammars.

4.3 Farshi's extension of Algorithm 1

The non-termination of Tomita's Algorithm 2 was first reported by Nozohoor-Farshi. In [NF91], Farshi attributes the problem of Algorithm 2 to the false assumption that only a finite number of nullable reductions can be applied between the shift of two consecutive input symbols. Instead of creating sub-frontiers for nullable reductions, Farshi introduces extra searching when a new edge is added from an existing node in the frontier of the GSS.

In this section we describe the operation of Farshi's correction and highlight the extra costs involved. The formal specification of the algorithm is taken from [NF91]. The layout and notation of the algorithm is similar to that of Tomita's, apart from the Reducer function which is renamed to Completer.

Farshi's recogniser

Variables

Γ: The parse graph.
U_i: The set of state vertices created just before shifting the input word a_{i+1}.
s_0: The initial state of the parser.
$\mathcal{A}$: The set of active nodes on which the parser will act.

$\mathcal{Q}$: The set of shift operations to be carried out.

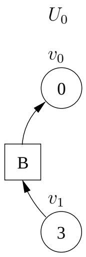
$\mathcal{R}$: The set of reductions to be performed.

```
function PARSE($G, a_1, ..., a_n$)
    $\Gamma := \emptyset$
    $a_{n+1} := \$$
    $r :=$ FALSE
    create a vertex $v_0$ labelled $s_0$ in $\Gamma$
    $U_0 := \{v_0\}$
    for $i := 0$ to $n$ do PARSEWORD($i$)
    return $r$

function PARSEWORD($i$)
    $\mathcal{A} := U_i$
    $\mathcal{R} := \emptyset$
    $\mathcal{Q} := \emptyset$
    repeat
        if $A \neq \emptyset$ then ACTOR
        else if $\mathcal{R} \neq \emptyset$ then COMPLETER
    until $\mathcal{R} = \emptyset$ and $\mathcal{A} = \emptyset$
    SHIFTER

function ACTOR
    remove an element $v$ from $\mathcal{A}$
    for all $\alpha \in$ ACTION(STATE($v$), $a_{i+1}$) do
        if $\alpha =$ 'accept' then $r =$ TRUE
        if $\alpha =$ 'shift $s$' then add $\langle v, s\rangle$ to $\mathcal{Q}$
        if $\alpha =$ 'reduce $p$' then
            for all vertices $w$ such that there exists a directed walk of length $2|\text{RHS}(p)|$
                from $v$ to $w$ do
                /* For $\epsilon$-rules this is a trivial walk, i.e. $w = v$ */
                add $\langle w, p\rangle$ to $\mathcal{R}$

function COMPLETER
    remove an element $\langle w, p\rangle$ from $\mathcal{R}$
    $N :=$ LHS($p$)
    $s :=$ GOTO(STATE($w$),$N$)
    if there exists $u \in U_i$ such that STATE($u$) $= s$ then
        if there does not exist a path of length 2 from $u$ to $w$ then
            create a vertex $z$ labelled $N$ in $\Gamma$
            create two arcs in $\Gamma$ from $u$ to $z$ and from $z$ to $w$
            for all $v \in (U_i - \mathcal{A})$ do
                /*In the case of non-$\epsilon$-grammars this loop executes for $v = u$ only*/
                for all $q$ such that ‘reduce $q$’ $\in$ ACTION(STATE($v$),$a_{i+1}$) do
                    for all vertices $t$ such that there exists a directed walk of length
                        $2|\text{RHS}(q)|$ from $v$ to $t$ that goes through vertex $z$ do
                        add $\langle t, q \rangle$ to $\mathcal{R}$
    else /* i.e., when there does not exist $u \in U_i$ such that STATE($u$) $= s$ */
        create in $\Gamma$ two vertices $u$ and $z$ labelled $s$ and $N$ respectively
        create two arcs in $\Gamma$ from $u$ to $z$ and from $z$ to $w$
        add $u$ to both $\mathcal{A}$ and $U_i$

function SHIFTER
    $U_{i+1} := \emptyset$
    repeat
        remove an element $\langle v, s \rangle$ from $\mathcal{Q}$
        create a vertex $x$ labelled $a_{i+1}$ in $\Gamma$
        create an arc from $x$ to $v$
        if there exists a vertex $u \in U_{i+1}$ such that STATE($u$) $= s$ then
            create an arc from $u$ to $x$
        else
            create a vertex $u$ labelled $s$ and an arc from $u$ to $x$ in $\Gamma$
            add $u$ to $U_{i+1}$
    until $\mathcal{Q} = \emptyset$
```

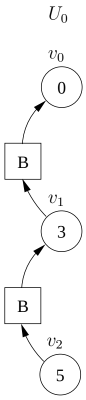
4.3.1 Example − a hidden-left recursive grammar

Consider the parse of the string *ba* (with Grammar 4.3) that caused Tomita's Algorithm 2 to fail to terminate in the previous section. We demonstrate how Farshi's algorithm deals with hidden-left recursive grammars by tracing the construction of the GSS.

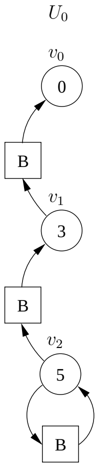
We create v_0, labelled by the start state of the DFA, in level U_0. Since there is a shift/reduce conflict in state 0, we queue the shift and perform the reduction on rule $B ::= \epsilon$. The reduction is nullable so we create the new state node, v_1 labelled 3, in the current level, and a path of length two from v_1 to v_0 via the new intermediate symbol node B.



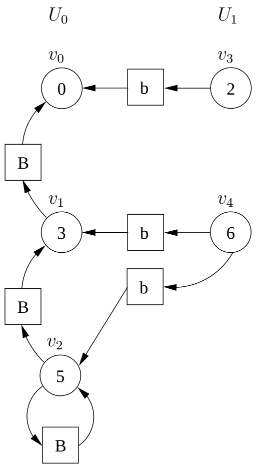
There is another shift/reduce conflict in state 3, so we queue the shift once again and perform the reduction on rule $B ::= \epsilon$. As a result we create v_2 in the current level and another symbol node labelled B. We make this new symbol node a successor of v_2 and a predecessor of v_1.



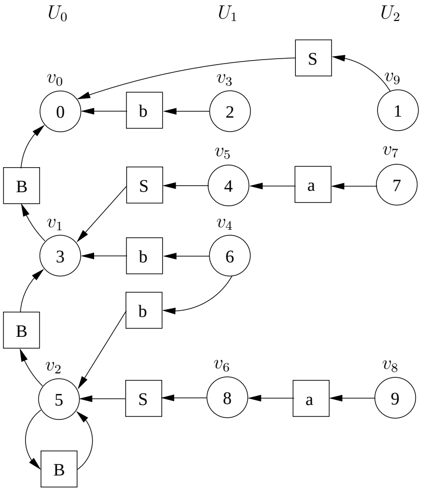
There is the same shift/reduce conflict in state 5, so we perform the nullable reduction and queue the shift action again. However, because of the loop in state 5 of the DFA, caused by the hidden-left recursive rule $S ::= BSa$, we create a cyclic path of length two from v_2 to itself via a new symbol node labelled B.



There are no further reductions that can be performed from any of the state nodes in level U_0, so we proceed to carry-out the queued shift actions. This results in two new state nodes, v_3 and v_4, being created in level U_1, with paths of length two, via three separate symbol nodes labelled b, back to the appropriate nodes in level U_0.



From this point on Farshi's algorithm behaves in the same way as Tomita's Algorithm 2. Once all actions have been performed the GSS shown below is constructed.



Since v_9 is labelled by the accept state of the DFA and we have consumed all the input string, we have successfully recognised *ba* as being in the language of Grammar 4.3.

4.3.2 Example – a hidden-right recursive grammar

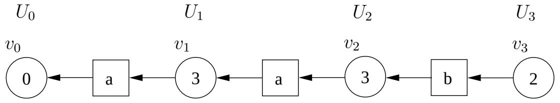
It is claimed that Farshi's algorithm "...works exactly like the original one [Tomita's] in case of grammars that have no ϵ-productions... [and] has no extra costs beyond that of the original algorithm." [NF91, p.74].

Although it is possible to implement the algorithm so as not to incur the extra searching costs for ϵ-free grammars the algorithmic description presented in [NF91] does not include any functionality to achieve it. The only indication that this is possible comes from the comment in the Completer function.

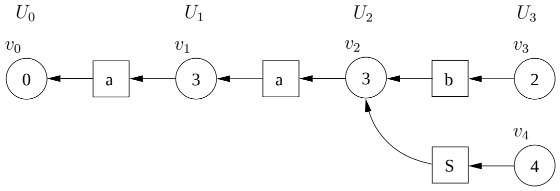
The extra searching costs introduced by Farshi's modification can be high if done naïvely. The algorithm, as described in [NF91], searches all nodes in the current frontier that are not waiting to be processed, and finds any reduction paths that pass through the new node created as part of the original reduction. This extra searching to correct Tomita's error for grammars containing right nullable rules effectively defeats the core of Tomita's idea – reduced graph searching. This fix can be thought of as a brute force way to fixing the problem.

To highlight the extra searching introduced by Farshi's algorithm we demonstrate its operation for the parse of the string *aab* with Grammar 4.2 shown on page 61. We begin by creating the start state, v_0, in U_0. We add v_0 to the set $\mathcal{A}$ and process it in the ACTOR. As there is only a shift to state 3, from state 0 on an a, $(v_0, 3)$ is added to $\mathcal{Q}$. We then create the state node v_1 in U_1, the symbol node labelled a and a path of length two from v_1 to v_0 via the symbol node.

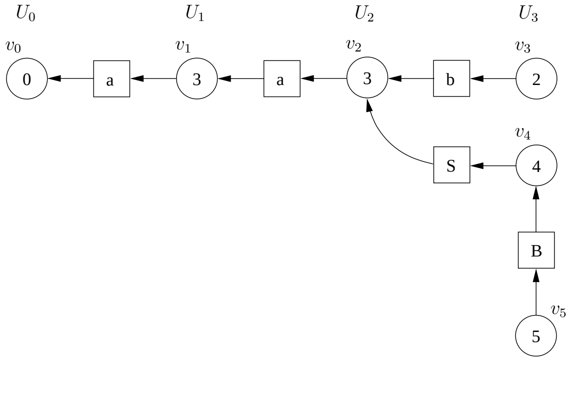
We parse the next two input symbols in this way, creating the two new state nodes v_2 and v_3.



When we process v_3 in the ACTOR we find the reduction on rule 2, $(S ::= b\cdot)$, in state 2. We trace back a path of length two from v_3 to v_2 and add $(v_2, 2)$ to $\mathcal{R}$. We process the contents of $\mathcal{R}$ in the COMPLETER which results in the new node, v_4, labelled 4, being created in U_3, with a path of length two to v_2 through a new symbol node labelled S.



Processing v_4 in the ACTOR we find an applicable reduction on rule 3, $(B ::= \cdot)$. Since the reduction is nullable, we do not traverse any edges and add $(v_4, 3)$ to the set $\mathcal{R}$. When we process $(v_4, 3)$ in the COMPLETER we create the new node, v_5, labelled 5, and add it to the set $\mathcal{A}$ and U_3. We create the symbol node labelled B and a path of length two from v_5 to v_4 via the new symbol node.

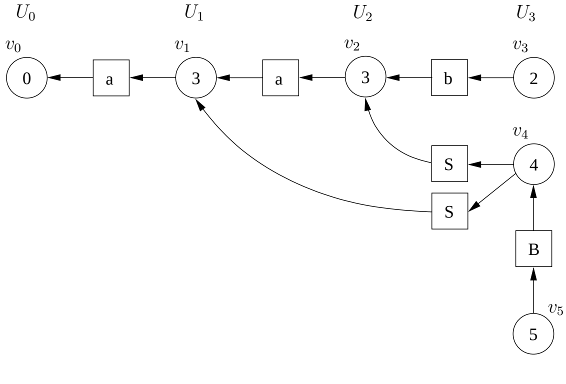


When we process v_5 we find the reduction on rule 1, $(S ::= aSB\cdot)$, in state 5. We trace back a path of length six to v_1 and add $(v_1, 1)$ to $\mathcal{R}$. When we process $(v_1, 1)$ in the COMPLETER, we find that there is already a node labelled 4 in the frontier of the GSS. As there is not a path of length two to v_1 from v_4, we create one via a new symbol node labelled S. However, because we have added a new edge to an existing node of the GSS we need to ensure that no new reduction paths have been introduced from the other nodes in the frontier. This is achieved by effectively traversing all reduction paths from the existing nodes in the frontier and re-performing any reductions that go through the new edge.

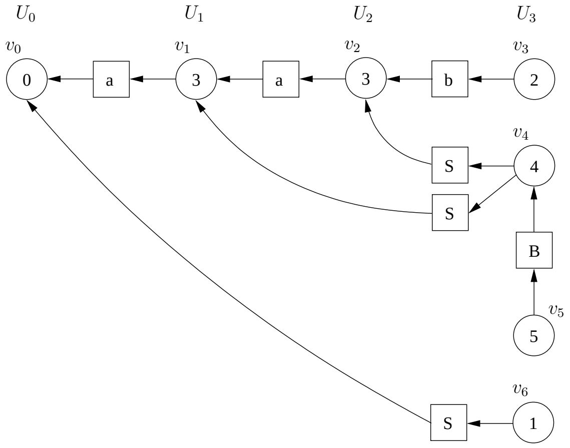
It is not necessary to re-trace reduction paths from nodes that are still waiting to be processed, as they will be carried out later. However, since there are no nodes in the set $\mathcal{A}$ at this point of our example, the set difference $U_3 - \mathcal{A}$, results in all nodes in U_3 being considered. We trace the paths:

- v_3, v_2 for reduction $(S ::= b\cdot)$;
- v_5, v_4, v_2, v_1 for reduction $(S ::= aSB\cdot)$;
- v_5, v_4, v_1, v_0 for reduction $(S ::= aSB\cdot)$.

Only the last traversal goes through the new edge between v_4 and v_1, so we add $(v_0, 1)$ to $\mathcal{R}$ for the reduction on rule 1.



When we perform the reduction in the COMPLETER we create the new node, v_6, labelled 1, and the symbol node labelled S with a path from v_6 to v_0. Since v_6 is labelled by the accept state of the DFA and we have consumed all the input string, the parse terminates in success. The final GSS constructed is shown below.



In practice this searching can trigger considerable extra work, see Chapter 10 where we discuss some experiments. It turns out that it is trivial to improve the efficiency of the algorithm by limiting the searching to paths within the current level. This is possible as the new path created from the reduction has to leave the current frontier. As soon as a path being searched goes to a previous level and does not go through the new edge, that particular search can be terminated.

We have implemented both the naïve and optimised versions of Farshi's algorithm. We compare both versions to each other in Chapter 10.

In the next section we discuss the construction of derivations by GLR parsers, specifically focusing on Tomita's approach and Rekers' modifications.

4.4 Constructing derivations

The GLR algorithm finds all possible parses of an input string. This is because Tomita was interested in natural language processing which often has "temporarily or absolutely ambiguous input sentences" and hence requires this approach to deal with it. Tomita recognised the problem of ambiguous parsing leading to exponential time requirements. The number of parses of a sentence with an ambiguous grammar may grow exponentially with the size of the sentence [CP82]. Therefore an efficient parsing algorithm would still require exponential time just to print the exponential number of possible parse trees. The key is to use an efficient representation of the parse trees. Tomita achieved this through subtree sharing and local ambiguity packing of the parse forest.

In this section we define the structure used by Tomita to represent all derivations of an input string and then discuss his parsing algorithm and Rekers' subsequent extension.

4.4.1 Shared packed parse forests

Local ambiguity occurs when there is a reduce/reduce conflict in the parse table. This makes it possible to reduce the same substring in more than one way which manifests itself in the parse tree as two or more nodes with the same label that have leaf nodes representing the same part of the input. A lot of local ambiguity can cause an exponential number of parse trees to be created. However, this can be controlled by combining all parse trees into one structure, taking advantage of sharing and packing of certain nodes, called a *Shared Packed Parse Forest* (SPPF). The parent nodes are merged into a new node and a packing node is made the parent of each of the subtrees.

For example, consider the ambiguous Grammar 4.4 (previously encountered on page 29), which defines the syntax of simple arithmetic expressions.

$$\begin{aligned} S' &::= E \\ E &::= E + E \mid E * E \mid a \end{aligned} \tag{4.4}$$

We can parse the string $a + a * a$ in two different ways that represents the left and right associativity of the $*$ symbol. The two parse trees are shown in Figure 4.4.

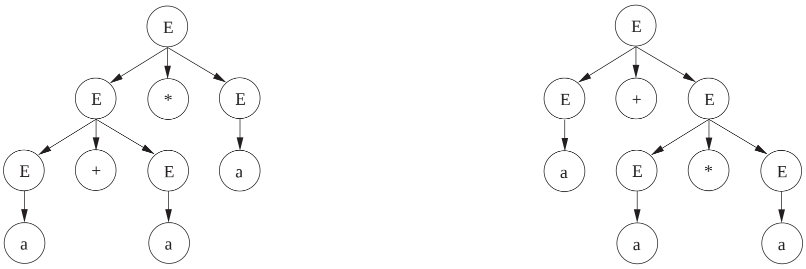


Figure 4.4: The two parse trees of the string $a + a * a$ for Grammar 4.4.

The amount of space required to represent both trees can be significantly reduced by using the SPPF shown in Figure 4.5.

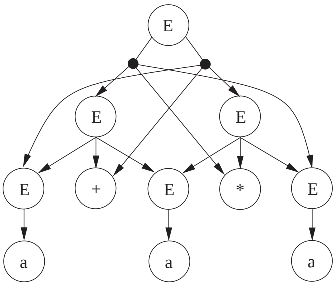


Figure 4.5: The SPPF for $a + a * a$ Grammar 4.4.

If two trees have the same subtree for a substring $a_j \dots a_i$ then that subtree can be shared, as is shown in Figure 4.6.

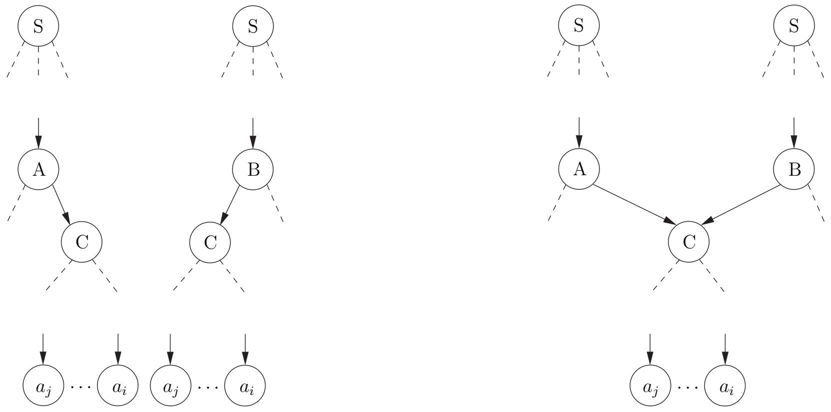


Figure 4.6: Subtree sharing.

If two trees have a node labelled by the same non-terminal that derives the same substring, $a_j \dots a_i$, in two different ways then that node can be packed and the two subtrees added as alternates under the newly packed node. For example, see Figure 4.7 below.

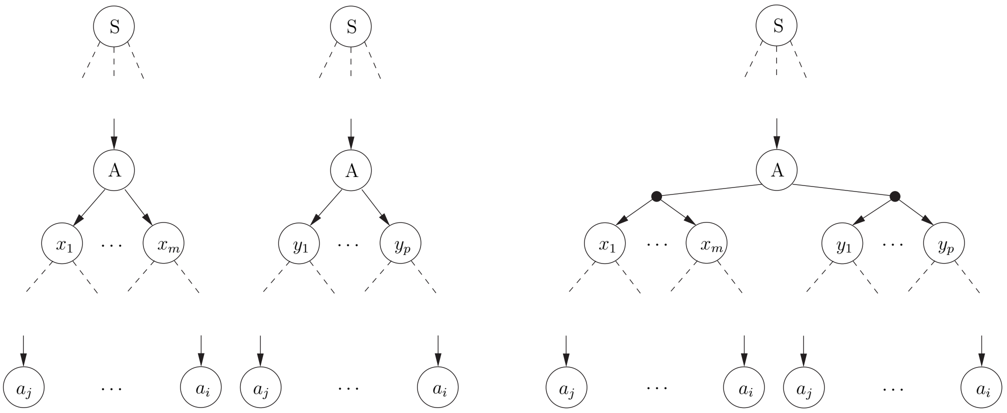


Figure 4.7: Use of packing nodes to combine different derivations of the same substring.

Although the SPPF provides an efficient representation of multiple parse trees it is not the most compact representation possible. It has been shown that maximal sharing can be achieved by "sharing the corresponding prefixes of right hand sides" [BL89]. This approach involves sharing all nodes that are labelled by the same symbol and derive terminals that are lexicographically the same. However, with this representation the yield of the derivation tree may not be the input string. This technique has been successfully adopted by the SGLR parser in the ASF+SDF Meta-Environment [vdBvDH^{+}01]. A description of the ATerm library, the data structure that efficiently implements the maximally shared SPPF, is given in [vdBdJKO00].

Cyclic grammars

A cyclic grammar contains a nonterminal that can derive itself, $A \overset{*}{\Rightarrow} A$. Cyclic grammars can have an infinite number of derivations for certain strings in their language. For example, the parse of the string a with Grammar 4.5 results in the construction of an infinite number of parse trees of the form shown in Figure 4.8.

$$\begin{aligned} S' &::= S \\ S &::= SS \mid a \mid \epsilon \end{aligned} \tag{4.5}$$

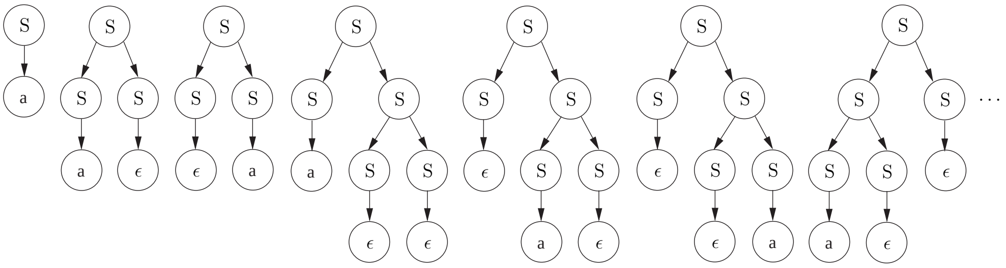


Figure 4.8: Some parse trees for Grammar 4.5 and input string a.

Farshi introduced cycles into the GSS so that his algorithm could be used to parse strings with cyclic grammars. We introduce cycles into the SPPF so that they can be used to represent an infinite number of parse trees. The SPPF representing the parse trees of the above parse is shown in Figure 4.9.

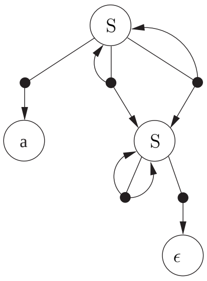


Figure 4.9: The SPPF for Grammar 4.5 and input string a.

4.4.2 Tomita's Algorithm 4

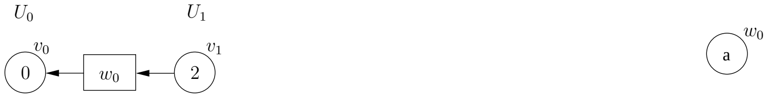
Tomita's Algorithm 3 is a recogniser, based on Algorithm 2, that incorporates sharing of symbol nodes into the GSS. Tomita extends Algorithm 3 to a parser, Algorithm 4, by introducing the necessary SPPF construction. Recall that Algorithm 2 can fail to terminate on hidden-left recursive grammars. This behaviour is inherited by Algorithm 4. We discuss the operation of Algorithm 4 to illustrate Tomita's approach to

SPPF generation.

The GSS's constructed by Tomita's algorithms contain exactly one symbol node between every connected pair of state nodes. Although the symbol nodes do not perform any specific function in the recognition algorithms, they play an important role in Algorithm 4 − they correspond directly to the nodes in the SPPF.

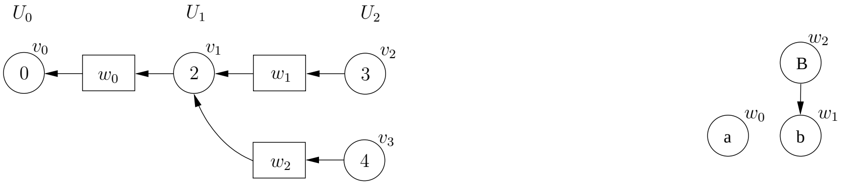
The amount of space required to represent all derivations of an ambiguous sentence is controlled by the sharing and packing of the nodes in the SPPF. Since the GSS symbol nodes have a one-to-one correspondence with the nodes in the SPPF, some of the symbol nodes in the GSS need to be shared. We demonstrate the construction of an SPPF for the string *abd* whose Grammar 4.1 and associated LR(1) DFA are shown on page 52.

We begin the parse by creating the node v_0, labelled by the start state of the DFA. The only applicable action from state 0 is a shift to state 2 for the first symbol on the input string, a. We perform the shift by creating the new state node, v_1, and the new SPPF node, w_0, labelled a. We use w_0 as the symbol node on the path from v_1 to v_0. To make the example easier to read we draw the SPPF separately from the GSS and only label the GSS symbol nodes with the node of the SPPF.

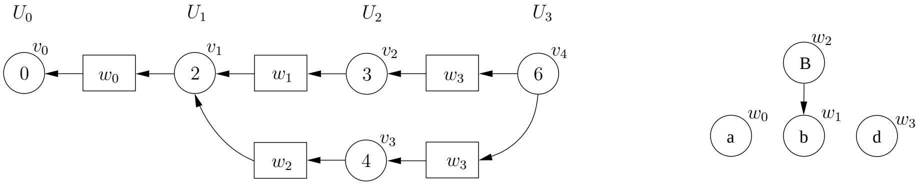


We continue the parse by processing v_1 and performing the associated shift to state 3. We create the new state node, v_2, labelled 3, and the new SPPF node, w_1, labelled b, which we use as the symbol node between v_1 and v_2.

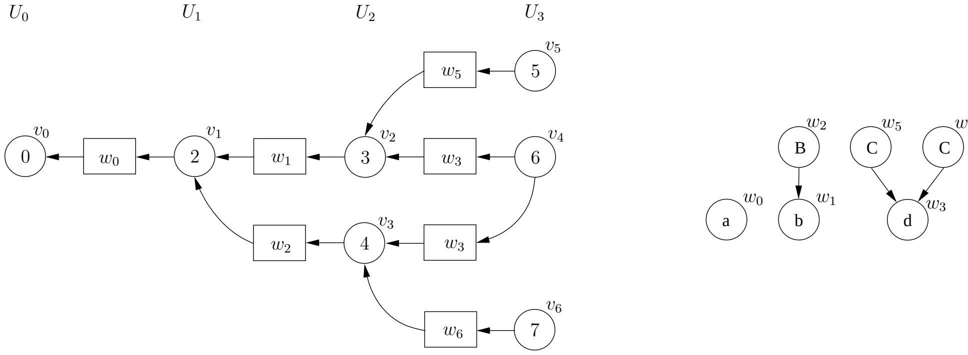
From state 3 there is a shift and a reduce action applicable. We queue the shift to state 6 and perform the reduction on rule 3, $B ::= b$. We begin by tracing back a path of length one from the symbol node w_1 to v_1. We then create a new state node, v_3, labelled 4 and the new SPPF node, w_2, labelled B. We make w_1 the child of w_2 and use w_2 as the symbol node on the path between v_3 and v_1. We process v_3 and queue the applicable shift action to state 6 which completes the construction of U_2.



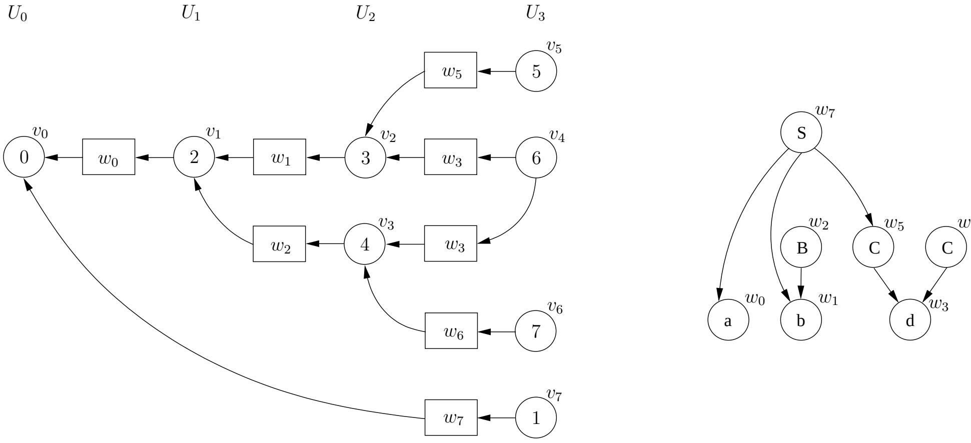
When we perform the two queued shift actions to state 6, we create a new state node, v_4, and a new SPPF node w_3, labelled d. We use w_3 as the symbol node on the path between both v_4 and v_2, and v_4 and v_3.



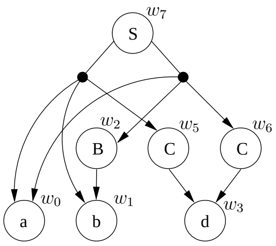
From state 6 there is a reduction on rule 4, $C ::= d$, applicable. We trace back the two separate paths of length one from the symbol nodes labelled w_3 which lead to v_2 and v_3 respectively. For the reduction path that leads to v_2 we create the new state node v_5, labelled 5, and the new SPPF node, w_5, labelled C. We make the symbol node, w_3, the child of w_5 and use w_5 as the symbol node between v_5 and v_2. For the second reduction path we create the state node v_6 and another SPPF node w_6, also labelled C with w_3 as its child.



Processing v_5 we find that the reduction on rule 1, $S ::= abC$, is applicable. We trace back a path of length five from w_5 to v_0 and collect the SPPF nodes w_1 and w_0 encountered on the traversal. We create the new state node v_7, labelled 1, and the new SPPF node, w_7, labelled S. We make w_0, w_1 and w_5 the children of w_7 and use w_7 as the symbol node on the path between v_7 and v_0.



Then we process v_6 and find the reduction on rule 2, $S ::= aBC$ that needs to be performed. We trace back a path of length five from w_6 to v_0 and collect the SPPF nodes w_2 and w_0. Because there is already the state node v_7 that is labelled 1, with an edge to v_0 we have encountered an ambiguity in the parse. Since the SPPF node, w_7, that is used as the symbol node on the path between v_7 and v_0 is labelled S and derives the same portion of input as this reduction, we can use packing nodes below w_7 to represent both derivations.



We have parsed all the symbols on the input string and since v_7 is labelled by the accept state of the DFA, the parse terminates in success.

Although the parse was successful the final SPPF is not as compact as possible; there are still several nodes that can be shared. Since the SPPF is encoded into the symbol nodes of the GSS, sharing can only be incorporated into the SPPF if the symbol nodes are shared in the GSS. Recall that two nodes in the SPPF can be shared if the are labelled by the same symbol and have the same subtree below them.

If a symbol node v has a parent node in level U_i and a child node in a level U_j then it derives $a_{j+1} \dots a_i$ of the input string $a_1 \dots a_n$ [SJ00]. Although such SPPF nodes can always be shared, it is not always possible to share the corresponding symbol node in the GSS. For example, if the two symbol nodes labelled A in the GSS below are shared, then two spurious reduction paths will be introduced

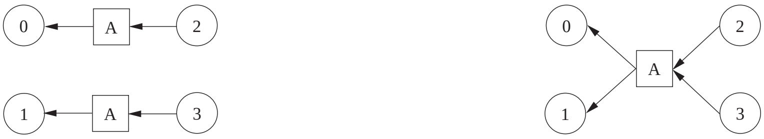


Figure 4.10: A GSS with incorrect symbol node sharing.

Although the increased sharing of symbol nodes reduces the size of the GSS and SPPF, it comes at a cost. Before creating a new symbol node with a parent node v, all symbol nodes directly linked to v must be searched to see if one reaches the same level as the one we want to create. So as to reduce the amount of searching performed in this way, Tomita only shares symbol nodes that are created during the same reduction. Specifically only symbol nodes that share the same reduction path up to the penultimate node are merged.

We shall now describe Rekers' parser version of Farshi's algorithm which generates more node sharing in the SPPF. It is this approach that we use in the RNGLR version of Tomita's algorithm described in Chapter 5.

4.4.3 Rekers' SPPF construction

It is often assumed that one of Rekers' contributions was a correction of Tomita's Algorithm 2. However, upon closer inspection it is clear that it is Farshi's and not Tomita's algorithm that forms the basis of Rekers' parser. Rekers' true contribution is the extension of Farshi's algorithm to a parser and the improved sharing of nodes in the SPPF.

Farshi only provides his algorithm as a recogniser, but he claims it is possible to extend it to a parser "...in a way similar to that of [Tomita]..." [NF91]. However, it is not straightforward to incorporate all the sharing as defined by Algorithm 3. For example, Farshi's algorithm finds the target of a reduction path as soon as a reduction is encountered which prevents the sharing of the non-terminal symbol nodes.

In order to achieve better sharing of nodes in the SPPF Rekers does not use symbol nodes in the GSS. Instead of having a one-to-one correspondence between the GSS and SPPF, Rekers labels each of the edges in the GSS with a pointer to the associated SPPF node. This enables more nodes in the SPPF to be shared without worrying about spurious reduction paths being introduced as a result.

To achieve this sharing, it is necessary to remember the SPPF nodes that are constructed at the current step of the algorithm. Recall that nodes in the SPPF can only be shared if they are labelled by the same symbol and derive (or cover) the same portion of the input string. A naïve approach to ensuring that a given node derives the same portion of the input would involve the traversal of its entire sub-tree. Rekers presents a more efficient approach that requires the SPPF nodes to be labelled by the start and end position of the string covered. This reduces the cost of the check to a comparison of the start and end positions.

Rekers' SPPF representation contains three types of node: symbol nodes, term nodes and rule nodes. The term nodes are labelled by terminal symbols and form the leaves of the SPPF. The symbol nodes are labelled by non-terminals and have edges to rule nodes. The rule nodes are labelled by grammar rules and are equivalent to Tomita's packing nodes. A major difference to Tomita's representation is that every symbol node has at least one rule node as a child, even if the symbol node only has one set of children. As a result the SPPF produced by Rekers for a non-ambiguous parse is larger than that produced by Tomita.

Rekers' parser with improved sharing in the SPPF

The formal description of Rekers' parsing algorithm is taken from [Rek92]. Although the core of Rekers' algorithm is similar to Farshi's, it uses slightly different notation. The sets *for-actor* and *for-shifter* are used instead of $\mathcal{A}$ and $\mathcal{Q}$ respectively and the set $\mathcal{R}$ is not used explicitly. Also, the functions GET-RULENODE, COVER, ADD-RULENODE and GET-SYMBOLNODE are added to achieve the increased sharing in the SPPF. Note that trees that derive ϵ do not cover any part of the input string. The function COVER handles this situation by always returning a non-empty position.

```
function PARSE($Grammar$, $a_1, ..., a_n$)
```

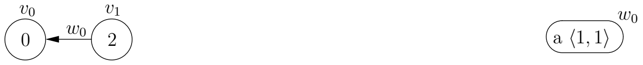
```
    $a_{n+1} = \$$
    **global** *accepting-parser* $:= \emptyset$
    create a stack node $p$ with state START-STATE(*Grammar*)
    **global** *active-parsers* $:= \{p\}$
    **for** $i := 1$ to $n+1$ **do**
        **global** *current-token* $:= a_i$
        **global** *position* $:= i$
        PARSEWORD
    **if** *accepting-parser* $\neq \emptyset$ **then**
        **return** the tree node of the only link of *accepting-parser*
    **else**
        **return** $\emptyset$

**function** PARSEWORD($i$)
    **global** *for-actor* := *active-parsers*
    **global** *for-shifter* $:= \emptyset$
    **global** *rulenodes* $:= \emptyset$; **global** *symbolnodes* $:= \emptyset$
    **while** *for-actor* $\neq \emptyset$ **do**
        remove a parser $p$ from *for-actor*
        ACTOR($p$)
    SHIFTER

**function** ACTOR($p$)
    **for all** *action*$\in$ ACTION(state($p$),*current-token*) **do**
        **if** *action* = (shift *state*$'$) **then**
            add $\langle p, state' \rangle$ to *for-shifter*
        **else if** *action* = (reduce $A ::= \alpha$ **then**)
            DO-REDUCTIONS($p, A ::= \alpha$)
        **else if** *action* = accept **then**
            *accepting-parser* := $p$

**function** DO-REDUCTIONS($p, A ::= \alpha$)
    **for all** $p'$ for which a path of length($\alpha$) from $p$ to $p'$ exists **do**
        *kids* := the tree nodes of the links which form the path from $p$ to $p'$
        REDUCER($p'$, GOTO(state($p'$),$A$), $A ::= \alpha$, *kids*)

**function** REDUCER($p^-$, *state*, $A ::= \alpha$, *kids*)
    *rulenode* := GET-RULENODE($A ::= \alpha$, *kids*)
    **if** $\exists p \in$ *active-parsers* with state($p$) = *state* **then**
        **if** there already exists a direct link *link* from $p$ to $p^-$ **then**
```

```
            ADD-RULENODE(tree node(*link*), *rulenode*)
        else
            $n$ := GET-SYMBOLNODE($A$, *rulenode*)
            add a link *link* from $p$ to $p^-$ with tree node $n$
            for all $p' \in$ (*active-parsers* $-$ *for-actor*) do
                for all (reduce *rule*) $\in$ ACTION(state($p'$), *current-token*) do
                    DO-LIMITED-REDUCTIONS($p'$, *rule*, *link*)
    else
        create a stack node $p$ with state *state*
        $n$ := GET-SYMBOLNODE($A$, *rulenode*)
        add a link from $p$ to $p^-$ with tree node $n$
        add $p$ to *active-parsers*
        add $p$ to *for-actor*

function DO-LIMITED-REDUCTIONS($p, A ::= \alpha$, *link*)
    for all $p'$ for which a path of length($\alpha$) from $p$ to $p'$ through *link* exists do
        *kids* := the tree nodes of the links which form the path from $p$ to $p'$
        REDUCER($p'$, GOTO(state($p'$),$A$), $A ::= \alpha$, *kids*)

function SHIFTER
    *active-parsers* := $\emptyset$
    create a term node $n$ with token *token* and cover $\langle$ *position*, *position* $\rangle$
    for all $\langle p^-$, *state'*$\rangle \in$ *for-shifter* do
        if $\exists p \in$ *active-parsers* with state($p$) = *state'* then
            add a link from $p$ to $p^-$ with tree node $n$
        else
            create a stack node $p$ with state *state'*
            add a link from $p$ to $p^-$ with tree node $n$
            add $p$ to *active-parsers*

function GET-RULENODE($r$, *kids*)
    if $\exists n \in$ *rulenodes* with rule($n$) = $r$ and elements($n$) = *kids* then
        return $n$
    else
        create a rule node $n$ with rule $r$, elements *kids* and cover COVER(*kids*)
        add $n$ to *rulenodes*
        return $n$

function COVER(*kids*)
    if *kids* = $\emptyset$ or $\forall$ *kid* $\in$ *kids:* COVER(*kid*) = empty then
```

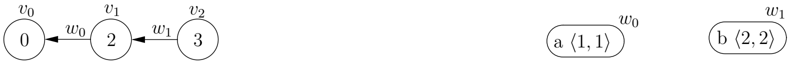
```
        return empty
    else
        begin := the start position of the first kid with a non-empty cover
        end := the end position of the last kid with a non-empty cover
        return ⟨ begin, end ⟩

function Add-Rulenode(symbolnode, rulenode)
    if rulenode ∉ the possibilities of symbolnode then
        add rulenode to the possibilities of symbolnode

function Get-Symbolnode(s, rulenode)
    if ∃n ∈ symbolnodes with symbol(n) = s and Cover(n) = Cover(rulenode)
        then
        Add-Rulenode(n, rulenode)
        return n
    else
        create a symbol node n with symbol s, possibilities { rulenode } and cover
            Cover(rulenode)
        add n to symbolnodes
        return n
```

We demonstrate the operation of Rekers' algorithm on the parse of the string *abd* whose Grammar 4.1 and associated LR(1) DFA are shown on page 52.

To begin the parse we create the new state node v_0, labelled by the start state of the DFA, and add it to *active-parsers*. We set the *current-token* to *a* and the *position* to 1. Then we execute the Parseword function, which leads us to copy v_0 into the *for-actor* and process it in the Actor function. Since there is shift to state 2 from state 0 in the DFA we add $\langle v_0, 2\rangle$ to *for-shifter*. We then perform the shift action in the Shifter which results in a new SPPF node w_0, labelled a $\langle 1, 1\rangle$ being created and used to label the edge between the new node v_1 and v_0.



We continue by setting the *current-token* to *b* and the *position* to 2 and then follow the same process as before to shift the input symbol *b*.

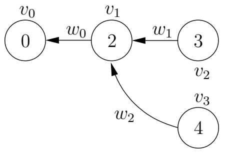


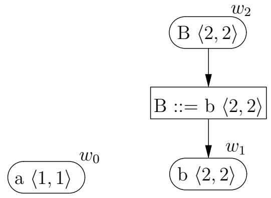
When we come to process v_2 in the Actor we find a shift/reduce conflict associated with state 3 on the current lookahead symbol *d*. We add $\langle v_2, 6\rangle$ to *for-shifter*, but before applying the shift, we perform the reduction on rule $B ::= b$ by executing the Do-Reductions function.

We trace a path of length one from v_2 to v_1, collecting the SPPF node that labels the edge traversed, and then execute the REDUCER function. In the REDUCER we use the GET-RULENODE function to find any existing rule nodes in the SPPF that derive the same portion of the input string. Since there is none, the new rule node labelled $B ::= b\ \langle 2, 2\rangle$ is created. We continue by creating the new GSS node v_3 labelled 4.

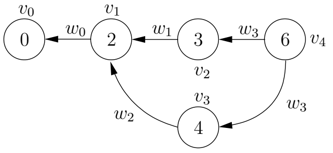
Before we create a new symbol node in the SPPF we execute the GET-SYMBOLNODE function to ensure that a node does not already exist that can be shared. No node is found so we create the new node w_2 and use it to label the edge between v_3 and v_1.

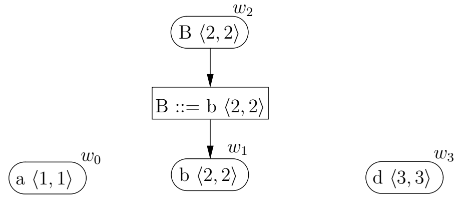
To ensure that the new node v_3 is processed we add it to *active-parsers* and *for-actor*.





Processing v_3 in the ACTOR function we find another shift action to state 6. We add $\langle v_3, 6\rangle$ to *for-shifter* and then proceed to perform the two queued shifts in the SHIFTER function. This results in the creation of the new GSS node v_4 labelled 6, with two edges back to v_2 and v_3 labelled with a pointer to the new SPPF node w_3.

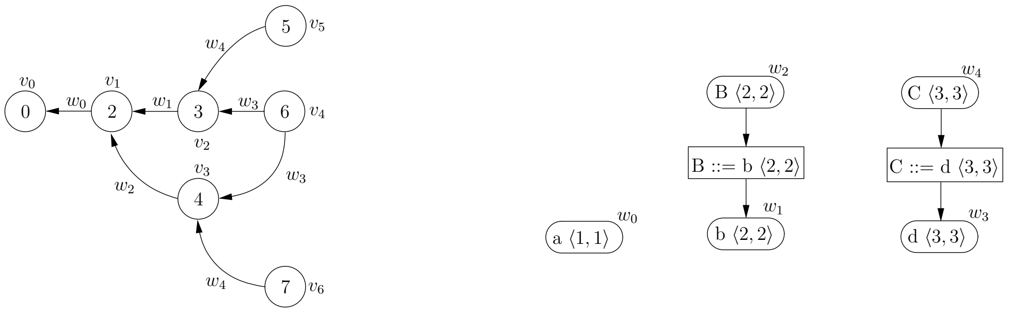




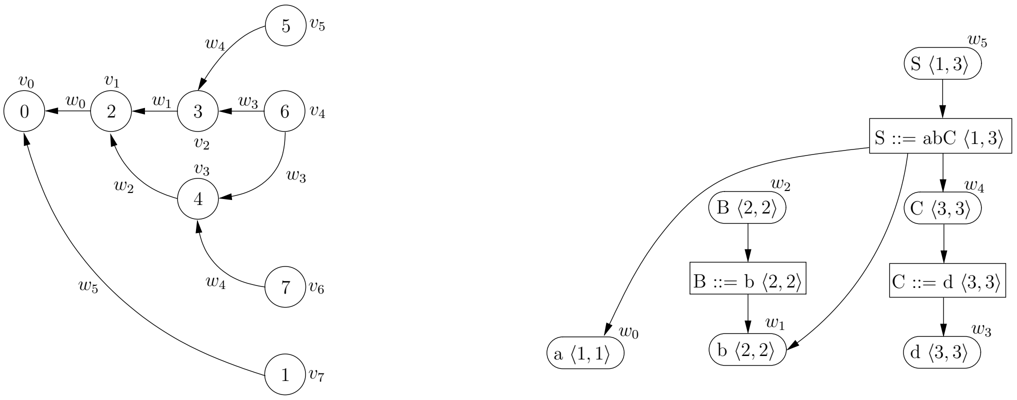
Processing v_4 in the ACTOR function we find a reduction on rule $C ::= d$. There are two different paths of length one that can be traced back from v_4 in the DO-REDUCTIONS function. For each path we collect the SPPF node that labels the edge traversed and execute the REDUCER function.

In the first execution of the REDUCER we create a new rule node labelled $C ::= d\ \langle 3, 3\rangle$ using the GET-RULENODE function and then create the new GSS node, v_5, labelled 5. Since there does not already exist an SPPF symbol node labelled $C\ \langle 3, 3\rangle$ we create w_4 and use a pointer to it to label the edge between v_5 and v_2. We also add v_5 to the *active-parsers* and *for-actor* sets.

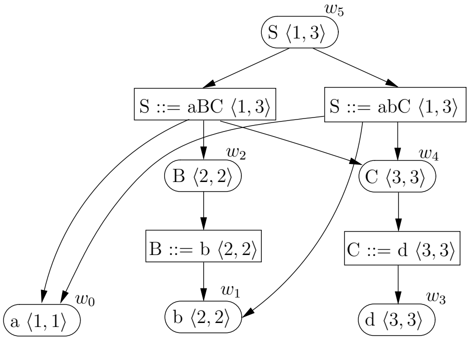
In the second execution of the REDUCER we find the rule node labelled $C ::= d\ \langle 3, 3\rangle$ that has the same set of children as the reduction we are performing. We create the new GSS node v_6, labelled 7 and pass the existing rule node into the GET-SYMBOLNODE function. However, since there is already the symbol node w_4 that is labelled by the same non-terminal and derives the same portion of the input, we do not create a new node. We use a pointer to w_4 to label the new edge between v_6 and v_3.



At this point the *for-actor* contains the two GSS nodes v_5 and v_6. Processing the first of the nodes we find the reduction on rule $S ::= abC$. We trace back a path of length three from v_5 and collect the SPPF nodes w_4, w_1, w_0 that label the edges traversed. We then create a new rule node labelled $S ::= abC \langle 1,3\rangle$ and make it the parent of the nodes previously collected. We create the new node v_7, labelled 1, and the new symbol node w_5, labelled $S \langle 1,3\rangle$, as a parent of the new rule node created. We use a pointer to w_5 to label the edge between v_7 and v_0 and add v_7 to the *active-parsers* and *for-actor* sets.



When we process v_6 we find another reduction on rule $S ::= aBC$. We trace back a path of length three to v_0 and collect the SPPF nodes w_0, w_2, w_4 that label the edges traversed. We create the new rule node labelled $S ::= aBC \langle 1,3\rangle$ and make it the parent of the previously collected SPPF nodes. However, we do not create a new state node in the GSS since v_7 is labelled by the goto state of the current reduction. The existence of an edge from v_7 to v_0 indicates an ambiguity in the parse. As a result we add the rule node created for this reduction as a child of w_5.



The only action associated to the state that labels v_7 is the accept action. Since all the input has been consumed the parse terminates in success and returns w_5 as the root of the SPPF.

It is clear that Rekers' SPPF construction produces more compact SPPF's than Tomita's Algorithm 4. This is because Rekers' algorithm is able to share more symbol nodes in the GSS than Tomita's algorithm. However, to achieve this extra sharing more searching is required which causes a significant overhead in the parsing process.

4.5 Summary

In this chapter we have introduced Tomita's GLR parsing technique. The recognition Algorithms 1 and 2 were discussed in detail. We demonstrated the construction of the GSS using Algorithm 1, and illustrated how a straightforward extension to deal with grammars containing ϵ-rules fails to parse certain grammars correctly. The operation of Algorithm 2 was then examined and its non-termination for hidden-left recursive grammars was demonstrated. The extension of Algorithm 1 due to Farshi was discussed in detail and the extra costs involved were highlighted.

Finally, Tomita's Algorithm 4 was presented that constructs an SPPF representation of multiple derivation trees and Rekers' extension of Farshi's algorithm was introduced.

Chapter 10 presents the experimental results of a naïve and optimised implementation of Farshi's recogniser for grammars which trigger worst case behaviour and several programming language grammars and strings.

The next chapter presents the RNGLR algorithm that correctly parses all context-free grammars using a modified LR parse table. The RNGLR parser incorporates some of Rekers' sharing into the SPPF, but reduces the amount of searching required.

Part II

Improving Generalised LR parsers

Chapter 5

Right Nulled Generalised LR parsing

Tomita's Algorithm 1 can fail to terminate when parsing strings in the language of grammars with hidden-right recursion and Algorithm 2 can fail on grammars with hidden-left recursion. In Chapter 4 we considered Farshi's extension to Tomita's Algorithm 1 that increases the searching required during the construction of the GSS, but which is able to parse all context-free grammars. This chapter presents an algorithm capable of parsing all context-free grammars without the increase in searching costs.

It is clear that Tomita was aware of the problems associated with Algorithm 1 as he restricted its use to ϵ-free grammars. It is not unusual to exclude ϵ-rules from parsing algorithms as difficulties are often caused by them; for instance the CYK algorithm requires grammars to be in Chomsky Normal Form. One of the reasons for using ϵ in grammars is that many languages can be defined naturally using ϵ-rules while the ϵ-free alternatives are often less compact and not as intuitive. Well known algorithms exist [AU73] that can remove ϵ from grammars, but parsers that depend on this technique build parse trees related to the modified grammar. Another approach [NS96] that performs automatic removal of ϵ-rules during a parse is discussed in Chapter 8.

This chapter presents a modification to the LR DFA's that allow Tomita's Algorithm 1 to parse all context-free grammars including those with ϵ-rules. We discuss Algorithm 1 on ϵ-rules in detail and examine some grammars that cause the parser to fail. A modification to the parse table is then given that causes the algorithm to work correctly. The RNGLR algorithm that parses strings using the modified parse table more efficiently than Algorithm 1 on ϵ-rules is then presented both as a recogniser and then as a parser. Finally, we discuss ways to reduce the extra non-determinism introduced by the modification made to the parse table.

5.1 Tomita's Algorithm 1e

Algorithm 1 provides a clear exposition of Tomita's ideas. Unfortunately it cannot be used to parse natural language grammars as they often contain ϵ-rules. Tomita's Algorithm 2 includes a complicated procedure for dealing with ϵ-rules which fails to terminate on grammars containing hidden-left recursion. The algorithm below, which we have called Algorithm 1e, is a straightforward extension to Algorithm 1 to include ϵ-rules, which works correctly for hidden-left recursive grammars, but may fail on grammars with hidden-right recursion. We begin this section by describing the modifications made to Algorithm 1 to allow ϵ-rules to be handled.

It is straight-forward to modify Tomita's Algorithm 1 so that it can handle grammars containing ϵ-rules. The two main changes that need to be made are in the Actor and Reducer functions. When an ϵ reduce action rX from a state v is found by the Actor, (v, X) is added to $\mathcal{R}$ instead of (u, X), where u are the successors of v. Since ϵ reductions do not require any states to be popped off the stack, no reduction paths are traced in the Reducer.

It has already been shown, in Chapter 4, that it is possible to add a new edge to an existing state in the current level of the GSS. In this case the reductions from the existing state need to be applied down the newly created reduction path. However, not all reductions need to be re-done. Since ϵ reductions do not pop any states off the stack, no new reduction path is added by the addition of the new edge. Because we store all pending reductions in the set $\mathcal{R}$, and all ϵ reductions are added to it when a new state is created, we can safely assume that if the ϵ reductions have not already been done, they eventually will. Therefore no ϵ reductions are added to $\mathcal{R}$ when a new edge is added to an existing state. This modification was added to Tomita's Algorithm 3, and since we have extended Algorithm 1 to allow ϵ grammars to be parsed we have included this modification to Algorithm 1e as well.

Tomita's original algorithms build GSS's with symbol nodes between state nodes. Algorithm 1e does not include the symbol nodes, but in our diagrams we shall label the edges to make the GSS's easier to visualise. As we shall see in Section 5.5 this approach also allows more compact SPPF's to be generated.

To allow a closer comparison to the algorithms we introduce later we have changed the layout of Algorithm 1e and the format of the actions in the parse table: instead of using sk and gk to represent the shift and goto actions, we use pk for both; instead of using rj to represent a reduction on rule j, we use $r(X, m)$ where X is the left hand non-terminal of rule j and m is the length of j's right hand side.

Algorithm 1e

```
input data start state $S_S$, accept state $S_A$, parse table $\mathcal{T}$, input string $a_1 ... a_n$

function PARSER($i$)
    create a node $v_0$ labelled $S_S$
    $U_0 = \{v_0\}, \mathcal{A} = \emptyset, \mathcal{R} = \emptyset, \mathcal{Q} = \emptyset, U_1 = \emptyset, ..., U_n = \emptyset$
    for $i = 0$ to $n$ do
        $\mathcal{A} = U_i$
        while $\mathcal{A} \neq \emptyset$ or $\mathcal{R} \neq \emptyset$ do
            if $\mathcal{A} \neq \emptyset$ then ACTOR($i$)
            else REDUCER($i$)
        SHIFTER($i$)
    if $S_A \in U_n$ then set $result = success$
    else $result = failure$
    return $result$

function ACTOR($i$)
    remove $v$ from $\mathcal{A}$
    let $h$ be the label of $v$
    if $pk \in \mathcal{T}(h, a_{i+1})$ then add $(v, k)$ to $\mathcal{Q}$
    for each $r(A, k) \in \mathcal{T}(h, a_{i+1})$ do
        if $k = 0$ then add $(v, A, k)$ to $\mathcal{R}$
        else for each successor node $u$ of $v$ do add $(u, A, k)$ to $\mathcal{R}$

function REDUCER($i$)
    remove $(v, X, m)$ from $\mathcal{R}$
    if $m = 0$ then
        let $k$ be the label of $v$ and let $pl \in \mathcal{T}(k, X)$
        if there is no node $w \in U_i$ labelled $l$ then
            create one and add it to $U_i$ and $\mathcal{A}$
        if there is not a path of length 1 from $w$ to $v$ then
            create an edge from $w$ to $v$
            if $w \notin \mathcal{A}$ then
                for all $r(B, t) \in \mathcal{T}(l, a_{i+1})$ with $t \neq 0$ do add $(v, B, t)$ to $\mathcal{R}$
    else
        for each node $u$ that can be reached from $v$ along a path of length $(m - 1)$
            do
            let $k$ be the label of $u$ and let $pl \in \mathcal{T}(k, X)$
            if there is no node $w \in U_i$ labelled $l$ then
            create one and add it to $U_i$ and $\mathcal{A}$
        **if** there is not a path of length 1 from $w$ to $u$ **then**
            create an edge from $w$ to $u$ labelled $X$
            **if** $w \notin \mathcal{A}$ **then**
                **for all** $r(B,t) \in \mathcal{T}(l, a_{i+1})$ with $t \neq 0$ **do** add $(u,B,t)$ to $\mathcal{R}$

**function** SHIFTER($i$)
    **while** $\mathcal{Q} \neq \emptyset$ **do**
        remove $(v,k)$ from $\mathcal{Q}$
        **if** there is no node $w \in U_{i+1}$ labelled $k$ **then** create one
        **if** there is not an edge labelled $a_{i+1}$ from $w$ to $v$ **then** create one
```

5.1.1 Example – a hidden-left recursive grammar

We revisit Example 4.2.6, which causes Algorithm 2 to fail to terminate, and show the operation of Algorithm 1e. We repeat Grammar 4.3 and use the LR(1) parse table below.

$$S' ::= S$$
$$S ::= BSa \mid b$$
$$B ::= \epsilon$$

	a	b	\$	B	S
0		p2/r(B,0)		p3	p1
1			acc		
2		r(S,1)			
3		p6/r(B,0)		p5	p4
4	s7				
5		p6/r(B,0)		p5	p8
6	r(S,1)				
7			r(S,3)		
8	p9				
9	r(S,3)				

We shall use Algorithm 1e to construct the GSS for the string *ba*. The start state v_0 labelled 0 is created in U_0 and the reduction $(v_0, 3)$ is added to the set $\mathcal{R}$ and $(v_0, 2)$ to the set $\mathcal{Q}$. Then $(v_0, 3)$ is removed from $\mathcal{R}$ which causes state v_1 labelled 3 and an edge (v_1, v_0) labelled B to be created and $(v_1, 3)$ to be added to $\mathcal{R}$ and $(v_1, 6)$

to $\mathcal{Q}$. When $(v_1, 3)$ is removed, state v_2 labelled 5 and an edge (v_2, v_1) labelled B are created and $(v_2, 3)$ is added to $\mathcal{R}$ and $(v_2, 6)$ to $\mathcal{Q}$. Removing $(v_2, 3)$ causes a cyclic edge to be created on state 5 also labelled B. The rightmost GSS in Figure 5.1, whose nodes have children within the same level, demonstrates the effect of ϵ-rules in the grammar. Notice the creation of the cycle in node v_2 that causes the algorithm to terminate in the case of hidden-left recursion.

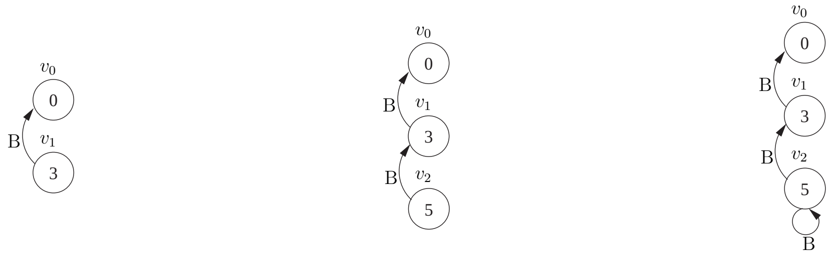
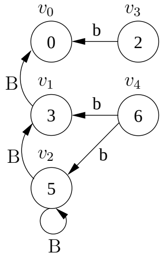
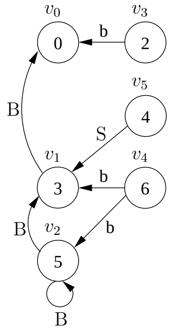


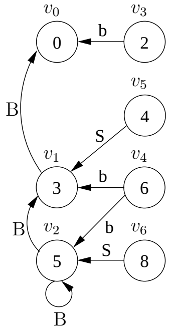
Figure 5.1: The GSS for Grammar 4.3 and input *ba* during the reductions in U_0.

At this point no other elements are in the sets $\mathcal{A}$ or $\mathcal{R}$ so we enter the SHIFTER which results in the following GSS being constructed.

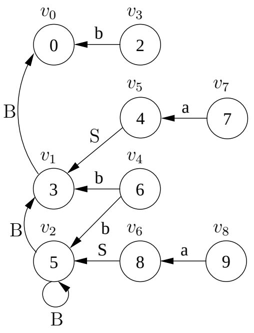


When the SHIFTER has completed, the set $\mathcal{A}$ contains the elements $\{v_3, v_4\}$. We remove v_3 and do nothing as there are no actions in $\mathcal{T}(2, a)$. When v_4 is removed and processed, $(v_1, 2)$ and $(v_2, 2)$ are added to $\mathcal{R}$ because of the reduction in $\mathcal{T}(6, a)$. When $(v_1, 2)$ is processed by the REDUCER, state v_5 labelled 4 and an edge (v_5, v_1) labelled S are created in U_1, and $(v_5, 7)$ is added to $\mathcal{Q}$. Then $(v_2, 2)$ is removed from $\mathcal{R}$ which results in a new state v_6 labelled 8 being created, with an edge (v_6, v_2) labelled S. The element $(v_6, 9)$ is also added to the set $\mathcal{Q}$.





As there are no other elements in $\mathcal{A}$ or $\mathcal{R}$ the Shifter is entered and the new states v_7 and v_8 labelled 7 and 9 are created in U_2 with edges (v_7, v_5) and (v_8, v_6) respectively.



As the set $\mathcal{A}$ is now $\{v_7, v_8\}$ the Actor removes and processes state v_7 which causes it to add $(v_5, 1)$ to $\mathcal{R}$. State v_8 is then removed, but it has no effect as there are no actions in $\mathcal{T}(9, \$)$. When $(v_5, 1)$ is processed by the Reducer a path of length 2 is traced back in the GSS to state v_0. The goto state in $\mathcal{T}(0, S)$ is 1, so the state v_9 labelled 1 with an edge (v_9, v_0) is created in U_2 and also added to the set $\mathcal{A}$. The Actor then removes v_9 and looks in $\mathcal{T}(1, \$)$ to find the accept state, which indicates that the parse succeeded and that the string *ba* is in the language of Grammar 4.3. Figure 5.2 shows the final GSS constructed by Algorithm 1e.

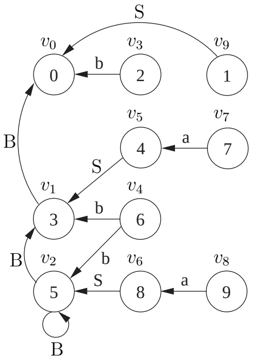


Figure 5.2: The final GSS for Grammar 4.3 and input *ba*.

5.1.2 Paths in the GSS

Algorithm 1 ensures that a reduction is only applied down a certain reduction path once. This is achieved by queueing the first edge of the path and the associated rule of the reduction in the set $\mathcal{R}$ when a new edge is created in the GSS. If a new edge is added to an existing node, any applicable reductions are applied down this new reduction path. For this approach to be successful it is essential that the new edge cannot be added to the middle of an existing reduction path. This is achieved by

Algorithm 1 since all new edges created have their source node in the frontier and their target in a previous level of the GSS. However, the modification of Algorithm 1e to deal with ϵ-rules allows edges to be created whose source and target nodes are in the same level. As a result of this, certain grammars can cause a new edge to be added to the middle of an existing reduction path. We illustrate this with the following example.

Example – an incorrect parse

Consider Grammar 5.1 and the associated LR(1) DFA in Figure 5.3 (previously encountered on page 61).

$$\begin{aligned} S' &::= S \\ S &::= aSB \mid b \\ B &::= \epsilon \end{aligned} \tag{5.1}$$

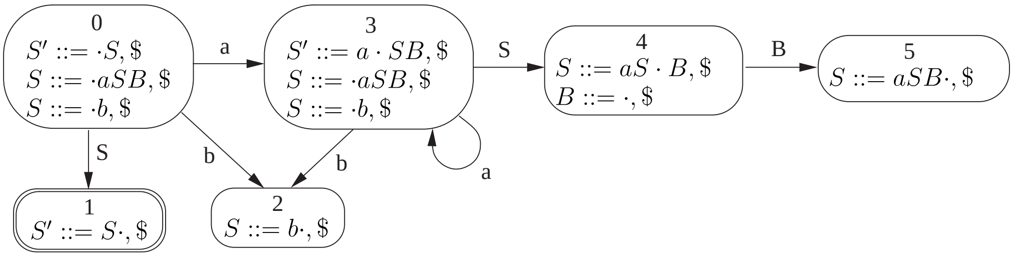


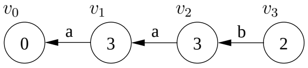
Figure 5.3: The LR(1) DFA for Grammar 5.1.

	a	b	$	B	S
0	p3	p2			p1
1			acc		
2			r(S,1)		
3	p3	p2			p4
4			r(B,0)	p5	
5			r(S,3)		

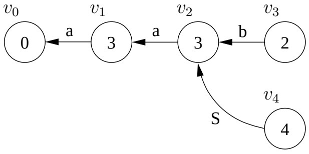
Table 5.1: The LR(1) parse table for Grammar 5.1.

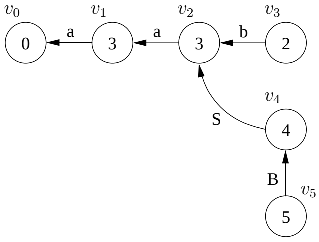
We shall use Algorithm 1e to construct the GSS for the string *aab*. We begin by creating the start state, v_0, in U_0 and look up the actions in $\mathcal{T}(0, a)$. As there is only a shift to state 3, $(v_0, 3)$ is added to $\mathcal{Q}$ and ultimately results in state v_1, and the edge (v_1, v_0) being created in U_1. The Actor then processes v_1 and adds $(v_1, 3)$ to $\mathcal{Q}$.

This results in a new state labelled 3 being created in U_2 and the edge (v_2, v_1) being added to the GSS. After processing v_2 the state v_3 labelled 2 and the edge (v_3, v_2) is created.



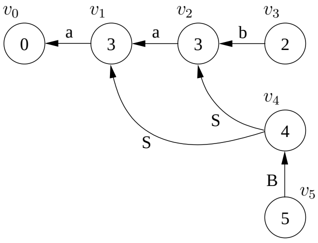
When the ACTOR processes v_3 it finds a reduce action in $\mathcal{T}(2, \$)$ which it adds to the set $\mathcal{R}$ as $(v_2, 2)$. The REDUCER then removes $(v_2, 2)$ and creates the state v_4 labelled 4 and the edge (v_4, v_2). Processing v_4 results in $(v_4, 3)$ being added to $\mathcal{R}$ which leads to the new state v_5 labelled 5 and the edge (v_5, v_4) labelled B being added to U_3.





When the new state v_5 is processed a reduce action is found in $\mathcal{T}(5, \$)$ which is added to $\mathcal{R}$ as $(v_4, 1)$. This reduction is traced back to v_1 whose goto state is 4. As a state already exists in U_3 labelled 4, a new edge (v_4, v_1) labelled S is added to the GSS. Because a new edge is added to an existing node, the reduction from v_4 is re-done, but as there is already a state labelled 5 and an edge (v_5, v_4) in the current level, nothing is done.

At this point all the sets are empty so U_3 is searched for the accept state, which is labelled 1. As no state labelled 1 exists in U_3 Algorithm 1e returns false as the result to the parse even though *aab* is in the language of Grammar 4.2.



It turns out that some grammars with right nullable rules (rules of the form $A ::= \alpha\beta$ where $\beta \overset{*}{\Rightarrow} \epsilon$) can be successfully parsed by Algorithm 1e when the ordering of reductions is carefully chosen. However, grammars such as 4.2 that contain hidden-right recursion will always fail to parse some sentences in their language [SJ00].

The next section presents a modification that can be made to the parse table, which will enable Algorithm 1e to correctly parse all context-free grammars.

5.2 Right Nulled parse tables

In order to allow Algorithm 1e to correctly parse all context-free grammars, a slightly modified parse table is built from the standard DFA in the following way. In addition to the standard reductions, we add reductions on right nullable rules, that is rules of the form $A ::= \alpha\beta$ where $\beta \overset{*}{\Rightarrow} \epsilon$.

If the DFA has a transition from state h to state k on the symbol a then pk is added to $\mathcal{T}(h, a)$ instead of sk or gk if a is a terminal or non-terminal respectively. If state h includes an item of the form $(A ::= x_1 \dots x_m \cdot B_1 \dots B_t, a)$ where $A \neq S'$ and $t = 0$ or $B_j \overset{*}{\Rightarrow} \epsilon$ for all $0 \leq j \leq t$, then $r(A, m)$ is added to $\mathcal{T}(h, a)$. If state h contains an item $(S' ::= S\cdot, \$)$ then acc is added to $\mathcal{T}(h, \$)$ and if $S' \overset{*}{\Rightarrow} \epsilon$ then acc is also added to $\mathcal{T}(0, \$)$. We call this type of parse table a Right Nulled (RN) parse table.

	a	b	$	B	S
0	p3	p2			p1
1			acc		
2			r(S,1)		
3	p3	p2			p4
4			r(B,0)/r(S,2)	p5	
5			r(S,3)		

Table 5.2: The RN parse table for Grammar 5.1.

Our approach takes advantage of the fact that no input is consumed when an ϵ is parsed. By performing the part of the right nullable reduction that does derive a portion of the input string we ensure that no new edges can be added to the middle of an existing reduction path. Clearly this prevents a reduction path being missed by a state in the sequence of nullable reductions.

It is possible to think that because nullable reductions do not consume any input it is safe not to apply them at all. Unfortunately this can cause problems which are discussed in detail in Section 5.3.

We note here that there are other parsing optimisations, such as [AH02], that have been proposed that require ϵ reductions to be taken account of at parser generation time. So if an item $A ::= \alpha \cdot B\beta \in U_i$ and $B \overset{*}{\Rightarrow} \epsilon$ the item $A ::= \alpha B \cdot \beta$ is added to U_i. This is not the same as the right nullable reductions in RN parse tables.

Although the RN parse table enables Algorithm 1e to work correctly for all context-free grammars, see [SJ00] for a proof of correctness, it contains more conflicts than its LR(1) counterpart. The next section presents a method to remove many of these conflicts introduced by the addition of right nullable reductions.

5.3 Reducing non-determinism

An undesirable side effect caused by the inclusion of the right nulled reductions in the RN parse table is a possible increase of non-determinism. As more reductions are added, it is possible that more reduce/reduce conflicts will occur (see Chapter 10).

Since our technique for correctly parsing hidden-right recursive grammars involves performing nullable reductions at the earliest point possible, one may hypothesise that the short circuited ϵ reductions can be removed from the table to reduce the non-determinism. Unfortunately this is not always possible because ϵ reductions may also include other useful derivations that do consume some input symbols.

For example, consider Grammar 5.2, the corresponding DFA in Figure 5.4 and the RN parse table 5.3.

$$\begin{aligned} S' &::= S \\ S &::= Ad \\ A &::= aAB \mid aABd \mid b \\ B &::= \epsilon \end{aligned} \tag{5.2}$$

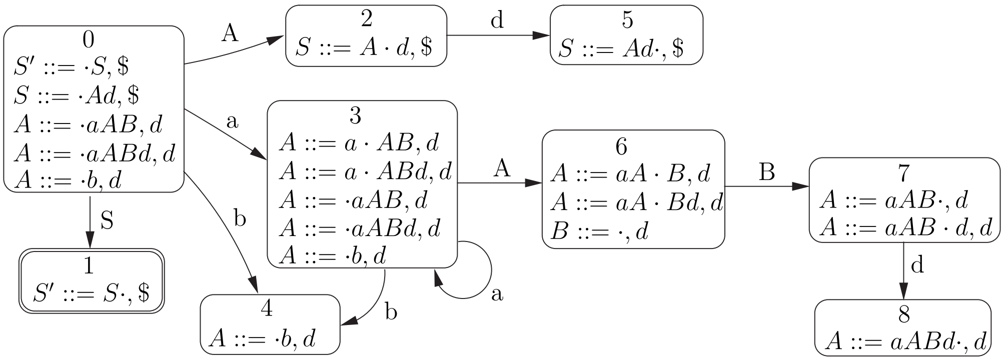


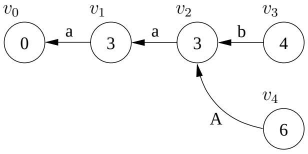
Figure 5.4: The LR(1) DFA for Grammar 5.2.

	a	b	d	$	A	B	S
0	p3	p4			p2		p1
1				acc			
2			p5				
3	p3	p4	r(A,1)		p6		
4			r(A,1)				
5				r(S,2)			
6			r(B,0)/r(A,2)				
7			p8/r(A,3)				
8			r(A,4)				

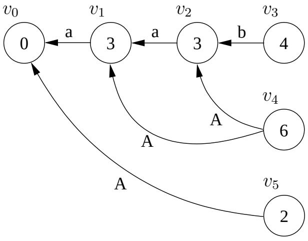
Table 5.3: The RN parse table for Grammar 5.2.

The LR(1) parse table is already non-deterministic, but the number of conflicts is increased when the RN reduction $r(A,2)$ is added to state 6. This will result in an increase in the amount of graph searching performed for certain parses such as the string *aabd*. Although the extra searching can be avoided by removing the ϵ reduction $r(B,0)$ from state 6, there are some strings that Algorithm 1 would then incorrectly fail to parse. For example, consider the parse of the string *aabdd* using parse table 5.3 and Algorithm 1.

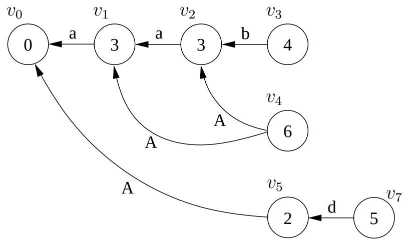
Once we have parsed the first three input symbols and performed the reduction $r(A,1)$ from state v_3 we have the following GSS.



Performing the RN reduction $r(A,2)$ from state v_4 causes the new edge between v_4 and v_1 to be created. This new edge introduces a new reduction path from v_4 to v_0, so we create v_5 and add an edge between it and v_0.



From state 2 there is a shift to state 5, so we create the new state node v_7 and the edge between v_7 and v_5.



By not performing the ϵ reduction from state 6 we avoid some redundant graph searching, but for this parse, we also incorrectly reject a string in the language of Grammar 5.2.

Although it is not generally possible to remove ϵ reductions from the parse table it is possible to identify points in the GSS construction when their application is redundant. The next section presents a general parsing algorithm, based on Algorithm 1e, that incorporates this allowing right nullable grammars to be parsed more efficiently. In Section 5.6 we discuss an approach that eliminates redundant reductions without compromising the correctness of the underlying parser for RN parse tables of LR grammars.

5.4 The RNGLR recognition algorithm

This section presents the Right Nulled GLR (RNGLR) recogniser which correctly parses all context-free grammars with the use of an RN parse table [SJ00]. A description of the algorithm is given and a discussion highlighting the differences between RNGLR and Algorithm 1e is undertaken.

RNGLR recogniser

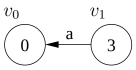
```
input data start state $S_S$, accept state $S_A$, RN table $\mathcal{T}$, input string $a_1...a_n$

function PARSER
    set $result = failure$
    if $n = 0$ then
        if acc $\in \mathcal{T}(S_S, \$)$ then set $result = success$
    else
        create a node $v_0$ labelled $S_S$
        set $U_0 = \{v_0\}, \mathcal{R} = \emptyset, \mathcal{Q} = \emptyset, a_{n+1} = \$, U_1 = \emptyset, ..., U_n = \emptyset$
        if $pk \in \mathcal{T}(S_S, a_1)$ then add $(v_0, k)$ to $\mathcal{Q}$
        for all $r(X, 0) \in \mathcal{T}(S_S, a_1)$ do add $(v_0, X, 0)$ to $\mathcal{R}$
        for $i = 0$ to $n$ do
            while $U_i \neq \emptyset$ do
                while $\mathcal{R} \neq \emptyset$ do REDUCER$(i)$
            SHIFTER($i$)
        if $S_A \in U_n$ then set $result = success$
    return $result$

function REDUCER($i$)
    remove $(v, X, m)$ from $\mathcal{R}$
    find the set $\chi$ of nodes which can be reached from $v$ along a path of length
        $(m-1)$, or length 0 if $m = 0$
    for each node $u \in \chi$ do
        let $k$ be the label of $u$ and let $pl \in \mathcal{T}(k, X)$
        if there is a node $w \in U_i$ with label $l$ then
            if there is not an edge from $w$ to $u$ then
                create an edge from $w$ to $u$
                if $m \neq 0$ then
                    for all $r(B,t) \in \mathcal{T}(l, a_{i+1})$ where $t \neq 0$ do add $(u, B, t)$ to $\mathcal{R}$
        else
            create a node $w \in U_i$ labelled $l$ and an edge from $w$ to $u$
            if $ph \in \mathcal{T}(l, a_{i+1})$ then add $(w, h)$ to $\mathcal{Q}$
            for all $r(B, 0) \in \mathcal{T}(l, a_{i+1})$ do add $(w, B, 0)$ to $\mathcal{R}$
            if $m \neq 0$ then
                for all $r(B,t) \in \mathcal{T}(l, a_{i+1})$ where $t \neq 0$ do add $(u, B, t)$ to $\mathcal{R}$

function SHIFTER($i$)
    if $i \neq n$ then
        set $\mathcal{Q}^{'} = \emptyset$
        while $\mathcal{Q} \neq \emptyset$ do
            remove an element $(v, k)$ from $\mathcal{Q}$
            if there is a node $w \in U_{i+1}$ with label $k$ then
                create an edge from $w$ to $v$
                for all $r(B,t) \in \mathcal{T}(k, a_{i+2})$ where $t \neq 0$ do add $(v, B, t)$ to $\mathcal{R}$
            else
                create a node $w \in U_{i+1}$ labelled $k$ and an edge from $w$ to $v$
                if $ph \in \mathcal{T}(k, a_{i+2})$ then add $(w, h)$ to $\mathcal{Q}^{'}$
                for all $r(B, 0) \in \mathcal{T}(k, a_{i+2})$ do add $(w, B, 0)$ to $\mathcal{R}$
                for all $r(B,t) \in \mathcal{T}(k, a_{i+2})$ where $t \neq 0$ do add $(v, B, t)$ to $\mathcal{R}$
        copy $\mathcal{Q}^{'}$ into $\mathcal{Q}$
```

The RNGLR algorithm looks very similar to Algorithm 1e but includes some subtle changes that make a big difference to the efficiency of the algorithm. One of the major differences in the appearance between the two algorithms is the lack of the ACTOR function in RNGLR. When a new node is created in the GSS by Algorithm 1e the ACTOR is used to perform the parse table lookup to retrieve the actions associated to the state that labels the new node. If a reduction of length > 0 is found then the ACTOR finds the new node's successors and adds the triple $\langle v, x, p\rangle$ to the set $\mathcal{R}$. The reason for using the successors of a new node is to ensure that when a reduction is done, it is only performed down the same path at most once. However, the successor nodes are known when a new node is created by the SHIFTER or REDUCER and by waiting to process a new node in the ACTOR, the information is lost and the algorithm needs to perform an unnecessary search for every node created. In comparison the RNGLR algorithm adds reductions to the set $\mathcal{R}$ when a new edge is created between two nodes. This results in the RNGLR algorithm performing one edge traversal less than Algorithm 1e for every reduction added to $\mathcal{R}$ when a new node is created.

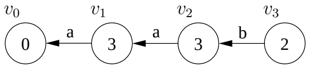
The RNGLR algorithm is optimised to exploit the features of an RN parse table. It takes advantage of the RN reductions to parse grammars with right nullable rules more efficiently than Algorithm 1e. Although one might expect the algorithm to perform more reductions and hence more edge visits as a result of the increased number of conflicts in the RN table, it turns out that fewer edge visits are performed because as we now discuss, reductions are not performed in certain cases.

If $\beta \overset{*}{\Rightarrow} \epsilon$ the RN parse table causes Algorithm 1e to do a reduction for a rule $A ::= \alpha \cdot \beta$ after it shifts the final symbol in α. So $|\beta|$ fewer edge visits are done for the reduction than for the rule $A ::= \alpha\beta\cdot$. However, the RN table also includes $|\beta|$ extra reductions, one for each nullable non-terminal in β, which Algorithm 1e also performs, more than eliminating the initial saving that was made. To prevent this from happening the RNGLR algorithm only adds new reductions to $\mathcal{R}$ if the length of the reduction that results in the new reduction path being created is greater than 0. The reasoning behind this is that if a new edge is created as part of a reduction of length 0, all reductions of length greater than 0 will have already been done by a previous RN reduction. For proofs of correctness of this approach see [SJ00].

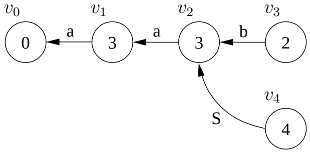
For example, consider the RNGLR parse of the string aab, with Grammar 4.2 and the RN parse table shown in Table 5.2. We begin by creating the state node v_0, labelled by the start state 0, in U_0. Since the only action in $\mathcal{T}(0, a)$ is a shift to state 3, we add $(v_0, 3)$ to $\mathcal{Q}$ and proceed to execute the SHIFTER. We create the new node v_1, labelled 3, in U_1 and lookup its associated actions in $\mathcal{T}(3, a)$. There is only a shift action to state 3, so $(v_1, 3)$ is first added to $\mathcal{Q}'$ and then to $\mathcal{Q}$.



We continue the parse by shifting the next two input symbols and creating the nodes v_2 and v_3 in U_2 and U_3 respectively. Upon the creation of v_3 in the SHIFTER we encounter a reduction $r(S, 1)$ in $\mathcal{T}(2, \$)$ and add $(v_2, S, 1)$ to $\mathcal{R}$.

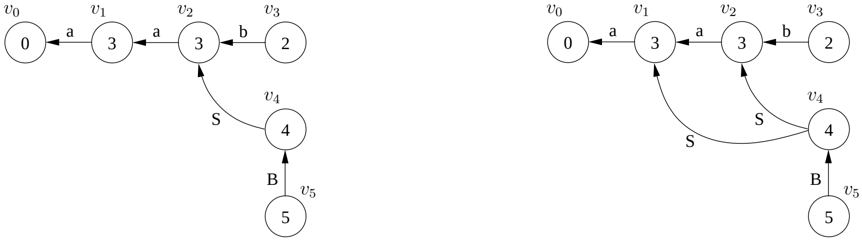


Processing $(v_2, S, 1)$ in the Reducer we find the shift to state 4 in $\mathcal{T}(3, \$)$. Since there is no node labelled 4 in U_3 we create v_4 and a new edge from v_4 to v_2. There is a reduce/reduce conflict in $\mathcal{T}(4, \$)$ so we add $(v_4, B, 0)$ and $(v_2, S, 2)$ to the set $\mathcal{R}$.

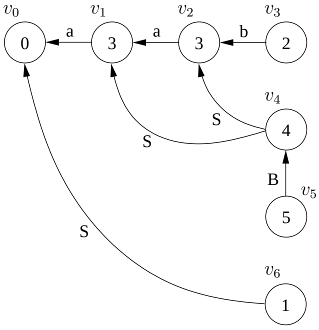


When $(v_4, B, 0)$ is processed by the Reducer the new node v_5, labelled 5, and the edge from v_5 to v_4 are created. Although there is a reduction in $\mathcal{T}(5, \$)$ it is not added to $\mathcal{R}$. This is a feature of the RNGLR algorithm to prevent reductions that are redundant from being performed. Since the right-nullable reduction $r(S, 2)$ was added to $\mathcal{T}(4, \$)$ the reduction that would normally be performed after all nullable non-terminals have been reduced was performed early.

We continue the parse by processing $(v_2, S, 2)$, and tracing back a path of length one from v_2 to v_1. Since there is already node labelled 4 in U_3 the new edge between v_4 and v_1 is created. The new edge creates a new reduction path that the reduction $r(S, 2)$ in state 4 can be applied down so $(v_1, S, 2)$ is added to $\mathcal{R}$.



Processing $(v_1, S, 2)$ we trace back a path of length one from v_1 to v_0 which results in the new state v_6 labelled 1 and the edge (v_6, v_0) being added to the GSS. All of the sets are now empty and since U_3 contains a state labelled by the DFA's accept state, the parse succeeds.



5.5 The RNGLR parsing algorithm

The parser version of the RNGLR algorithm is a straightforward extension of the recogniser described in the previous section. Chapter 4 presented both Tomita's and Rekers' approach to the construction of the SPPF. This section provides a brief overview of the main points in that chapter and a discussion of the approach taken by the RNGLR parser.

There are three conflicting goals associated with the construction of an SPPF: building a compact structure; minimising the amount of time taken to build the structure; and ensuring that for ambiguous sentences one derivation tree can be efficiently extracted. Tomita focused on the efficiency of the construction, only implementing a minimal amount of sharing and packing, thereby increasing the space required for the SPPF. Rekers extended Tomita's approach by increasing the amount of SPPF node sharing at the cost of introducing more searching to the algorithm. The RNGLR parser implements Rekers' approach in conjunction with several techniques designed to reduce the amount of searching required by the algorithm.

Rekers splits the SPPF nodes into two categories: nodes for the non-terminals called rule nodes and nodes for the terminals called symbol nodes. In order to achieve the sharing in the SPPF the rule and symbol nodes created for the current level are stored in two distinct sets. Before either type of node is created, the required set is searched for an existing node that covers the same part of the input. To do this without having to inspect all the subtrees of a node, the start and end positions of the input that the particular node derives are also stored with the node. A leaf node created for the input symbol at position i, is labelled (a, i, i). A rule node is labelled in a similar way, but takes the first value from its leftmost child and the second value from its rightmost child. So for the input $a_1 \dots a_d$, a rule node labelled (X, j, i) means that $X \overset{*}{\Rightarrow} a_j \dots a_i$. If $X \overset{*}{\Rightarrow} \epsilon$ then $i, j = 0$.

The RNGLR parser implements Rekers' approach for the construction of the SPPF as opposed to Tomita's because of the more compact SPPF built. Although both the RNGLR and Rekers' algorithm build the same SPPF they do so in a slightly different way. Instead of storing the SPPF nodes for the terminals and non-terminals separately, the RNGLR algorithm uses one set called $\mathcal{N}$ which is reset after each iteration in PARSER. In addition to this, because we always know the current level being constructed, only the start position of the string derived is included in an SPPF node. So instead of labeling an SPPF node with a triple (X, j, i), it is labelled with the pair (X, j) and stored in $\mathcal{N}$.

The RNGLR algorithm only works when used in conjunction with an RN parse table. When a grammar containing right nullable rules is parsed, the right nullable reductions will be done without tracing back over the edges containing the SPPF nodes for the nullable non-terminals. As a result it is necessary to build the SPPF trees for the nullable non-terminals, and the rightmost nullable strings of non-terminals of a right night nullable rule, before the parse is begun. These nullable trees are called ϵ-SPPF trees and since they are constant for a given grammar, they can be constructed when the parser is built and included in the RN parse table. Instead of storing reductions as the tuple $r(X, m)$, the RN parse table stores the triple $r(X, m, f)$, where X is a non-terminal, m is the length of the reduction and f is an index into a function I that returns the root of the associated ϵ-SPPF tree. If no ϵ-SPPF tree is associated with such a reduction then $f = 0$.

If all the ϵ-SPPF trees are created for all the nullable reductions, the final SPPF will not be as compact as it could be. So the ϵ-SPPF's are only constructed for nullable non-terminals and nullable strings γ, such that $|\gamma| > 1$ and there is a grammar rule of the form $A ::= \alpha\gamma$, where $\alpha \neq \epsilon$. Nullable strings like γ are called the *required nullable parts*. For rules of the form $A ::= \gamma$ where $\gamma \overset{*}{\Rightarrow} \epsilon$ the ϵ-SPPF for A is used instead.

In order to create the index to the ϵ-SPPF trees it is necessary to go through the grammar and, starting at one, index the required nullable parts and the non-terminals that derive ϵ. Before constructing the ϵ-SPPF trees, create the node u_0 labelled ϵ. Then create the ϵ-SPPF trees with the root node $u_{I(\omega)}$, labelled ω, for the nullable non-terminals or required nullable parts ω. In the RN parse table for a reduction $(A ::= \alpha \cdot \gamma, a)$ write $r(A, m, f)$, where $|\alpha| = m$ and $f = I(\gamma)$ if $m \neq 0$ and $f = I(A)$ if $m = 0$.

The elements added to the sets $\mathcal{Q}$ and $\mathcal{Q}'$ are the same as those used by the RNGLR recogniser, but the elements added to $\mathcal{R}$ contain more information. When a new edge is added between two nodes v and w, any applicable reductions from v are added to the set $\mathcal{R}$. For a reduction of the form $r(X, m, f)$, where $m > 0$, we add (w, X, m, f, z) to the set $\mathcal{R}$. The first three elements, w, X, m are the same as those used in the recogniser (state node, non-terminal, length of reduction). If the reduction is an RN-reduction of the form $(X ::= \alpha \cdot \beta)$ then f is the index into the function I which stores the root node of the ϵ-SPPF for β and z is the SPPF node that labels the edge between v and w. If the length of the reduction is zero $(m = 0)$ then f is the index into I for the root of the ϵ-SPPF of X and z is ϵ, since the edge between v and w is not traversed.

For example, consider Grammar 5.3. The nullable non-terminals are B and C, and the required nullable parts are BBC and BC. We define $I(B) = 1$, $I(C) = 2$, $I(BC) = 3$ and $I(BBC) = 4$. The associated ϵ-SPPF is shown in Figure 5.5.

$$\begin{aligned} &S' ::= S \\ &S ::= aBBC \\ &B ::= b \mid \epsilon \\ &C ::= \epsilon \end{aligned} \tag{5.3}$$

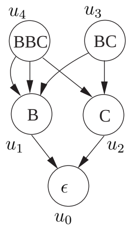


Figure 5.5: The ϵ-SPPF for Grammar 5.3.

Before presenting the formal specification of the RNGLR parser we demonstrate how the GSS and SPPF are constructed for the parse of the string *ab* in Grammar 5.3. The associated LR(1) DFA and RN parse table are shown in Figure 5.6 and Table 5.4.

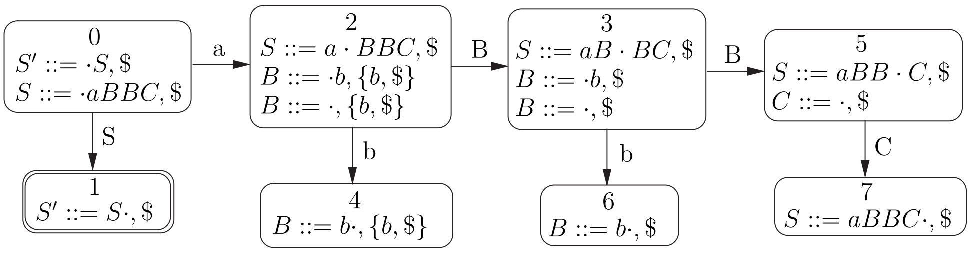


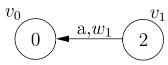
Figure 5.6: The LR(1) DFA for Grammar 5.3.

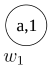
	a	b	$	B	C	S
0	p2					p1
1			acc			
2		p4/r(B,0,1)	r(B,0,1)/r(S,1,4)	p3		
3		p6	r(B,0,1)/r(S,2,3)	p5		
4		r(B,1,0)	r(B,1,0)			
5			r(C,0,2)/r(S,3,2)		p7	
6			r(B,1,0)		p7	
7			r(S,4,0)		p7	

Table 5.4: The LR(1) parse table for Grammar 5.3.

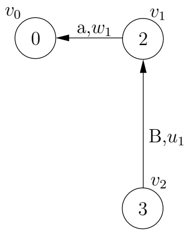
We create v_0, labelled by the start state of the DFA, and add it to U_0. Since the only applicable action in $\mathcal{T}(0, a)$ is a shift to state 2, we add $(v_0, 2)$ to $\mathcal{Q}$. The Shifter removes $(v_0, 2)$ from $\mathcal{Q}$ and creates a new SPPF node, w_1, labelled $(a, 1)$. Then since no node labelled 2 exists in the next level, v_1 is created and added to U_1, with an edge back to v_0 labelled by a and w_1. There is a shift/reduce conflict in $\mathcal{T}(2, b)$: a shift to state 4 and a reduction $r(B, 0, 1)$. We add $(v_1, 4)$ to $\mathcal{Q}'$ and $(v_1, B, 0, 1, \epsilon)$ to $\mathcal{R}$.

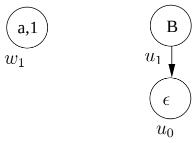
This completes the construction of the first level and the initialisation of U_1. The GSS and SPPF constructed up to this point are shown below.



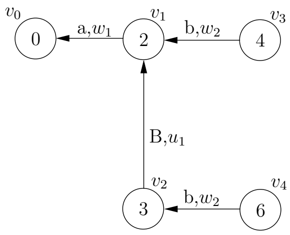


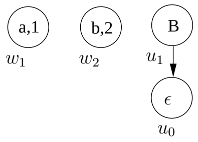
Next we process the reduction $(v_1, B, 0, 1, \epsilon)$ in the Reducer. It is a nullable reduction so we do not trace back a path in the GSS, but we do create the new node, v_2, labelled 3, with an edge back to v_1. We label the edge by the non-terminal of the reduction, B, and the root of the ϵ-SPPF tree for B. Since there is a shift action in $\mathcal{T}(3, b)$ we add $(v_2, 6)$ to $\mathcal{Q}$.





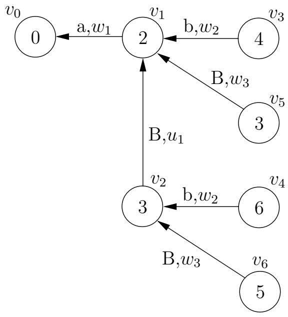
Processing the two queued shift actions in $\mathcal{Q}$ results in the construction of the SPPF node w_2 and the two new GSS nodes v_3 and v_4. There is a reduction $r(B, 1, 0)$ from both nodes, so we add $(v_1, B, 1, 0, w_2)$ and $(v_2, B, 1, 0, w_2)$ to $\mathcal{R}$.

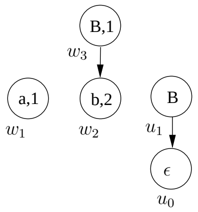




Processing the first of these reductions, we create the new SPPF node, w_3, labelled $(B, 1)$, add it to the set $\mathcal{N}$ and then use it to label the edge between the new GSS node, v_5, and v_1. We make the SPPF node, w_2, that labelled the edge between v_3 and v_1 the child of w_3. Since there is a reduce/reduce conflict, $r(B, 0, 1)$ and $r(S, 2, 3)$, in $\mathcal{T}(3, \$)$ we add $(v_5, B, 0, 1, \epsilon)$ and $(v_1, S, 2, 3, w_3)$ to $\mathcal{R}$.

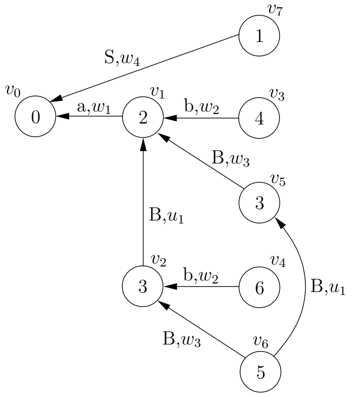
When we process the queued reduction $(v_2, B, 1, 0, w_2)$ we create the new GSS node v_6 labelled 5 and an edge to v_2. However, because there has already been the SPPF node, w_3, labelled $(B, 1)$ created in the current step of the algorithm (and stored in the set $\mathcal{N}$) we re-use it to label the edge between v_6 and v_2.

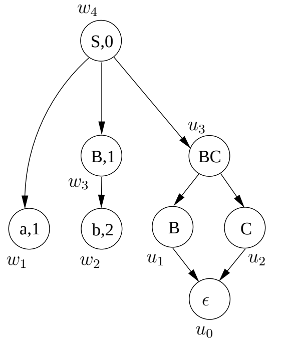




At this point $\mathcal{R} = \{(v_5, B, 0, 1, \epsilon), (v_1, S, 2, 3, w_3), (v_6, C, 0, 2, \epsilon), (v_2, S, 3, 2, w_3)\}$. We remove and process the first of these reductions which results in the new edge, labelled (B, u_1), being added between v_6 and v_5. Although the new edge has introduced a new reduction path from v_6, we do not add anything to the set $\mathcal{R}$ because the reduction performed was of length zero.

Next we process $(v_1, S, 2, 3, w_3)$. We trace back a path of length one from v_1 to v_0, collecting the SPPF node, w_1, that labels the edge traversed. We create the new GSS node, v_7, labelled 1 and an edge between v_7 and v_0. We create the new SPPF node, w_4, labelled $(S, 0)$ with edges pointing to the nodes w_1, w_3 and u_3 and use it to label the edge between v_7 and v_0 in the GSS. We also add w_4 into the set $\mathcal{N}$.





Processing the reduction encoded by $(v_6, C, 0, 2, \epsilon)$ we create the new GSS node v_8, labelled 7, and an edge from v_8 to v_6. Since the reduction is nullable, we label the new edge by the ϵ-SPPF node u_2 and do not add any other reductions to $\mathcal{R}$.

We then process the final reduction in $\mathcal{R}$, $(v_2, S, 3, 2, w_3)$. We trace back a path of length two from v_2 to v_0 and collect the SPPF nodes u_1 and w_1 that label the traversed edges. We search the set $\mathcal{N}$ for a node labelled $(S, 0)$ and find w_4. Since it does not have a sequence of children $[w_1, u_1, w_3, u_2]$ we create two new packing nodes below w_4 and add the existing children of w_4 to one and the new sequence to the other.

At this point all the input string has been parsed and no other actions remain to be processed. Since the accept state of the DFA labels v_7, the parse is successful and the root of the SPPF is the node that labels the edge between v_7 and v_0, w_4. The final GSS and SPPF constructed during the parse are shown in Figure 5.7.

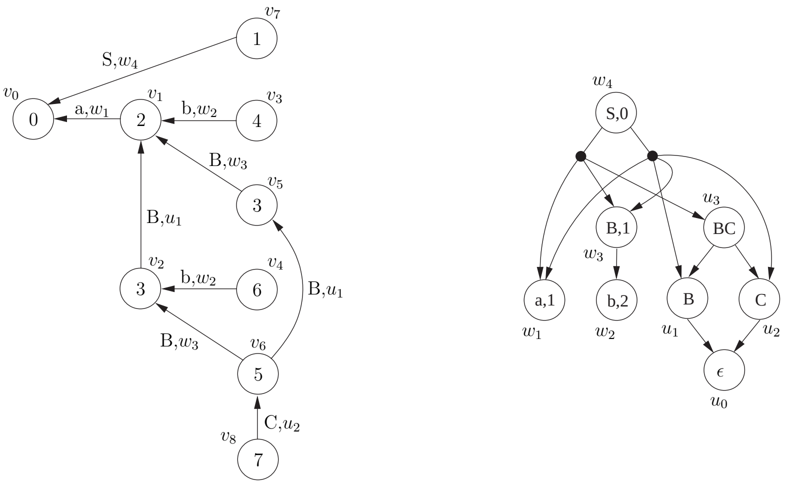


Figure 5.7: The final GSS and SPPF for the parse of ab in Grammar 5.3.

RNGLR parser

input data start state S_S, accept state S_A, RN table $\mathcal{T}$, input string $a_1...a_n$, the root nodes of the nullable SPPF's.

```
function PARSER
    set $result = failure$
    if $n = 0$ then
        if acc $\in \mathcal{T}(S_S, \$)$ then
            set $sppfRoot = u_{I(\omega)}$
            set $result = success$
    else
        create a node $v_0$ labelled $S_S$
        set $U_0 = \{v_0\}, \mathcal{R} = \emptyset, \mathcal{Q} = \emptyset, a_{n+1} = \$, U_1 = \emptyset, ..., U_n = \emptyset$
        if $pk \in \mathcal{T}(S_S, a_1)$ then add $(v_0, k)$ to $\mathcal{Q}$
        for all $r(X, 0, f) \in \mathcal{T}(S_S, a_1)$ do add $(v_0, X, 0, f, \epsilon)$ to $\mathcal{R}$
        for $i = 0$ to $n$ do
            while $U_i \neq \emptyset$ do
                $\mathcal{N} = \emptyset$
                while $\mathcal{R} \neq \emptyset$ do REDUCER($i$)
                SHIFTER($i$)
        if $S_A \in U_n$ then
            set $sppfRoot$ to the node that labels the edge $(S_A, v_0)$
            remove the SPPF nodes that are not reachable from $sppfRoot$
            set $result = success$
    return $result$
```

```
**function** Reducer($i$)
    remove $(v, X, m, f, y)$ from $\mathcal{R}$
    find the set $\chi$ of paths of length $(m-1)$ (or length 0 if $m = 0$) from $v$
    **if** $m \neq 0$ **then** let $w_m = y$
    **for each** path $\in \chi$ **do**
        let $w_{m-1}, ..., w_1$ be the edge labels and $u$ be the final node on the path
        let $k$ be the label of $u$ and let $pl \in \mathcal{T}(k, X)$
        **if** $m = 0$ **then** let $z = u_f$
        **else**
            suppose $u \in U_c$
            **if** there is no node $z \in \mathcal{N}$ labelled $(X, c)$ **then**
                create an SPPF node $z$ labelled $(X, c)$
                add $z$ to $\mathcal{N}$
        **if** there is a node $w \in U_i$ with label $l$ **then**
            **if** there is not an edge from $w$ to $u$ **then**
                create an edge from $w$ to $u$ labelled $z$
                **if** $m \neq 0$ **then**
                    **for all** $r(B, t, f) \in \mathcal{T}(l, a_{i+1})$ where $t \neq 0$ **do** add $(u, B, t, f, z)$ to $\mathcal{R}$
        **else**
            create a node $w \in U_i$ labelled $l$ and an edge from $w$ to $u$ labelled $z$
            **if** $ph \in \mathcal{T}(l, a_{i+1})$ **then** add $(w, h)$ to $\mathcal{Q}$
            **for all** $r(B, 0, f) \in \mathcal{T}(l, a_{i+1})$ **do** add $(w, B, 0, f, \epsilon)$ to $\mathcal{R}$
            **if** $m \neq 0$ **then**
                **for all** $r(B, t, f) \in \mathcal{T}(l, a_{i+1})$ where $t \neq 0$ **do** add $(u, B, t, f, z)$ to $\mathcal{R}$
        **if** $m \neq 0$ **then** AddChildren($z, w_1, ..., w_m, f$)

**function** Shifter($i$)
    **if** $i \neq n$ **then**
        set $\mathcal{Q}' = \emptyset$
        create an SPPF node $z$ labelled $(a_{i+1}, i)$
        **while** $\mathcal{Q} \neq \emptyset$ **do**
            remove $(v, k)$ from $\mathcal{Q}$
            **if** there is a node $w \in U_{i+1}$ with label $k$ **then**
                create an edge from $w$ to $v$ labelled $z$
                **for all** $r(B, t, f) \in \mathcal{T}(k, a_{i+2})$ where $t \neq 0$ **do** add $(v, B, t, f, z)$ to $\mathcal{R}$
            **else**
                create a node $w \in U_{i+1}$ labelled $k$ and an edge from $w$ to $v$ labelled $z$
                **if** $ph \in \mathcal{T}(k, a_{i+2})$ **then** add $(w, h)$ to $\mathcal{Q}'$
                **for all** $r(B, 0, f) \in \mathcal{T}(k, a_{i+2})$ **do** add $(w, B, 0, f, \epsilon)$ to $\mathcal{R}$
            for all $r(B,t,f) \in \mathcal{T}(k,a_{i+2})$ where $t \neq 0$ do add $(v,B,t,f,z)$ to $\mathcal{R}$
        copy $\mathcal{Q}'$ into $\mathcal{Q}$

function AddChildren($y, w_1, ..., w_m, f$)
    if $f = 0$ then let $\Lambda = (w_1, ..., w_m)$
    else let $\Lambda = (w_1, ..., w_m, u_f)$
    if $y$ has no children then
        for each node $\vartheta \in \Lambda$ do add an edge from $y$ to $\vartheta$
    else  if $y$ does not have a sequence of children labelled $\Lambda$ then
        if $y$ does not have a child which is a packing node then
            create a new packing node $z$ and an edge from $y$ to $z$
            add edges from $z$ to all other children of $y$
            remove all edges from $y$ apart from the one to $z$
        create a new packing node $t$ and an edge from $y$ to $t$
        for each node $\vartheta \in \Lambda$ do add an edge from $t$ to $\vartheta$
```

5.6 Resolvability

In Section 5.3 we discussed the effect of the extra non-determinism created as a result of the nullable reductions added to the RN parse table. In this section we present an approach, first described in [JS02], of removing redundant nullable reductions from the RN parse table of LR grammars with right nullable rules. In certain cases this new *resolved* parse table can be used by the standard LR parsing algorithm, to parse certain strings with less stack activity. It has been shown in [SJ03a] that for an LR grammar it is possible to remove all reduce/reduce conflicts from an RN parse table so that the standard LR parsing algorithm can be used to parse sentences with less stack activity.

In order to remove reductions from a state k without breaking the parser, it is necessary for k and a lookahead a to conform to the following two properties.

1. For each $a \in \mathbf{T} \cup \{\$\}$ there is at most one item $(X ::= \tau \cdot \sigma, a) \in k$, such that $\tau \neq \epsilon$ and $\sigma \overset{*}{\Rightarrow} \epsilon$.
2. If $(X ::= \tau \cdot \sigma, a) \in k$, where $\sigma \overset{*}{\Rightarrow} \epsilon$, and $(W ::= \alpha \cdot \beta, g) \in k$, where $a \in \textsc{first}(\beta)$, then $\tau = \epsilon$ and any derivation $\beta \overset{*}{\Rightarrow} au$ includes a step $Xau \overset{*}{\Rightarrow} \sigma au$.

Such states are called *a-resolvable*. The two properties above work on the principle that a state k with an item of the form $(X ::= \tau \cdot \sigma, a)$, where $\tau \neq \epsilon$ and $\sigma \overset{*}{\Rightarrow} \epsilon$ can have all but one of its reductions, for the lookahead a, removed as long as property 2 is not broken. This is because we can define the order in which rules are added to a DFA state and hence guarantee that the reduction for the rule $X ::= \tau\sigma$ must take place before any other action can happen.

The reduction that is not removed from an a-resolvable state k is known as the *base reduction* of k for lookahead a. To formally define a base reduction it is necessary to first define a function to calculate the order in which items are added to a DFA state.

DEFINITION 5.1 *Let k be a DFA state and let $(X ::= \tau \cdot \sigma, a)$ be an item in k. If $\tau = \epsilon$ then we define $level_k(X ::= \tau \cdot \sigma, a) = 0$. We also define $level_0(S' ::= \cdot S, \$) = 0$. For $X \neq S'$ and $\tau = \epsilon$ we let*

$$r = min\{level_k(Y ::= \gamma \cdot X\delta, b) \mid (Y ::= \gamma \cdot X\delta, b) \in k, a \in \text{FIRST}(\delta b)\}$$

and then define $level_k(X ::= \cdot\sigma, a) = (r + 1)$. [SJ03a]

DEFINITION 5.2 *Let Γ be any context-free grammar and let k be an a-resolvable state in the DFA for Γ. An item $(X ::= \tau \cdot \sigma, a) \in k$ is a base reduction on a in k if, for all other items $(Y ::= \gamma \cdot \delta, a) \in k$ such that $\delta \overset{*}{\Rightarrow} \epsilon, level_k(X ::= \tau \cdot \sigma, a) \leq level_k(Y ::= \gamma \cdot \delta, a)$.[SJ03a]*

Table 5.8 shows how the resolved RN parse table of the LR(1) grammar 5.4 can be used by the standard LR(1) parsing algorithm, to parse a sentence using less stack activity than when the LR(1) table is used.

$$\begin{aligned}S' &::= S\\ S &::= aBCD\\ B &::= \epsilon\\ C &::= \epsilon\\ D &::= \epsilon\end{aligned} \tag{5.4}$$

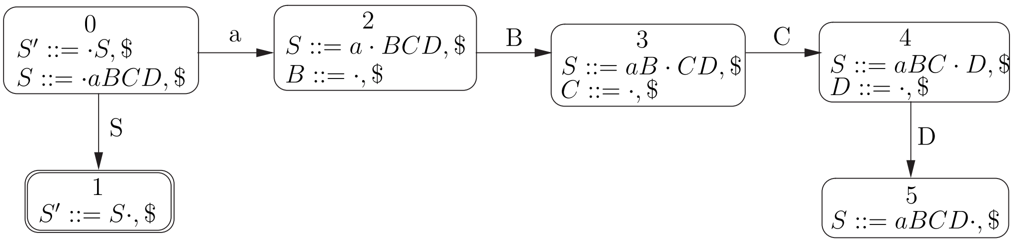


Figure 5.8: The LR(1) DFA for Grammar 5.4.

	a	$	B	C	D	S
0	p2					p1
1		acc				
2		r(B,0)	p3			
3		r(C,0)		p4		
4		r(D,0)			p5	
5		r(S,4)				

Table 5.5: The LR(1) parse table for Grammar 5.4.

	a	$	B	C	D	S
0	p2					p1
1		acc				
2		r(B,0)/r(S,1)	p3			
3		r(C,0)/r(S,2)		p4		
4		r(D,0)/r(S,3)			p5	
5		r(S,4)				

Table 5.6: The RN parse table for Grammar 5.4.

	a	$	B	C	D	S
0	p2					p1
1		acc				
2		r(S,1)	p3			
3		r(S,2)		p4		
4		r(S,3)			p5	
5		r(S,4)				

Table 5.7: The resolved RN parse table for Grammar 5.4.

LR(1)			Resolved RN		
Stack	Input	Action	Stack	Input	Action
$\$0$	a	s2	$\$0$	a	p2
$\$0a2$	$\$$	r2	$\$0a2$	$\$$	r(S,1)
$\$0a2B3$	$\$$	r3	$\$0S1$	$\$$	acc
$\$0a2B3C4$	$\$$	r4			
$\$0a2B3C4D5$	$\$$	r1			
$\$0S1$	$\$$	acc			

Table 5.8: Trace of LR(1) parse of input a for Grammar 5.4 using an LR(1) and a resolved RN parse table.

5.7 Summary

This chapter has presented Algorithm 1e − a straightforward extension of Tomita's Algorithm 1 that can deal with hidden-left recursion, but which fails to parse grammars containing hidden-right recursion. A modification to the standard LR(1) parse table was introduced which causes Algorithm 1e to correctly parse all context-free grammars. Tables containing this modification are called RN tables. The RNGLR recognition algorithm that parses all context-free grammars with the use of an RN parse table was described and its operation demonstrated with various examples. The RNGLR parser that constructs an SPPF in the style of Rekers, but which employs less searching, was also presented.

Chapter 10 presents the experimental results that abstract the performance of Algorithm 1e, Algorithm 1e mod, and the RNGLR algorithm for grammars which trigger worst case behaviour and several programming language grammars and strings.

The next chapter presents a cubic worst case general parsing algorithm called BRNGLR.

Chapter 6

Binary Right Nulled Generalised LR parsing

Despite the fact that general context-free parsing is a mature field in Computer Science, its worst case complexity is still unknown. The algorithm with the best asymptotic time complexity to date is presented in [Val75] by Valiant. His approach uses Boolean matrix multiplication (BMM) to construct a recognition matrix that displays the complexity of the associated BMM algorithm. Since publication of Valiant's paper, more efficient BMM algorithms have been developed. The algorithm with currently the lowest asymptotic complexity displays $O(n^{2.376})$ for $n \times n$ matrices [CW87, CW90].

Although the approach taken by Valiant is unlikely to be used in practice, because of the high constant overheads, it was an important step toward understanding more about the complexity of general context-free parsers. A related result has been presented by Lee which maps a context-free parser with $O(n^{3-\epsilon})$ complexity to an algorithm to multiply two $n \times n$ Boolean matrices in $O(n^{3-(\epsilon/3)})$ time. A side effect of this work has led to the hypothesis that 'practical parsers running in significantly lower than cubic time are unlikely to exist' [Lee02]. Although this analysis does suggest that linear time parsers are unlikely to exist, it does not preclude quadratic algorithms from being developed.

Two other general parsing algorithms that have been used in practice are the CYK and Earley algorithms. Both display cubic worst case complexity, although the CYK algorithm requires grammars to be transformed to CNF before parsing. Unfortunately this complexity is still too high for certain applications; most programming languages have largely deterministic grammars which can be parsed by linear parsers. The GLR algorithm first developed by Tomita, provides a general parsing algorithm that takes advantage of the efficiency of the deterministic LR parsers. Unfortunately these algorithms display unbounded polynomial time and space complexity [Joh79].

This chapter presents an algorithm which is based on the RNGLR algorithm described in Chapter 5, but which has a worst case complexity of $O(n^3)$ without requiring any transformations to be done to the grammar.

6.1 The worst case complexity of GLR recognisers

In Chapter 2 we examined the operation of a standard bottom-up shift reduce parser. Recall that such a parser works by using a stack to collect symbols of a sentential form and then when it recognises a handle, of length m say, it pops the m symbols off the stack and pushes on the left hand non-terminal of the corresponding rule. An important feature of such shift reduce parsers is that it is not necessary to examine the symbols on the stack before they are popped. The associated automaton guarantees that when a reduction for a rule $A ::= \beta$ is performed, the top m symbols on the stack are equal to β. Implementations of such parsers exploit this property and perform reductions in unit time by simply decrementing the stack pointer. In comparison, general parsing algorithms that extend such parsers cannot perform reductions as efficiently.

It is well known that the worst case time complexity of Tomita's GLR parser is $O(n^{M+1})$, where n is the length of the input string and M is the length of the longest grammar rule [Joh79]. Although it is not entirely surprising, it is somewhat disappointing that the recogniser displays the same complexity, especially since both the Earley and CYK recognisers are cubic in the worst case. In this section we shall explain why the RNGLR recogniser is worse than cubic.

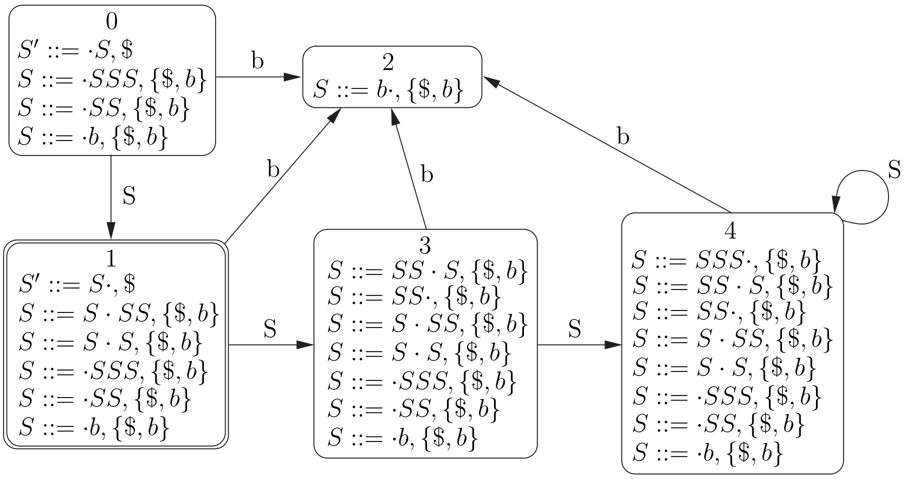
Roughly, when a GLR parser reaches a non-deterministic point in the automaton, the stack is split and both parses are followed. If the separate stacks then have a common stack top once again, the stacks are merged. This merging of stacks bounds the size of the GSS to at most $(n + 1) \times H$ nodes, where n is the length of the input string and H is the number of states in the automaton. However, because the different stacks are now combined into one structure a reduce action can be applied down several paths from one node. Unlike the standard shift reduce parser we are not able to simply decrement a stack pointer, we need to perform a search.

It is possible for a state in level i to have edges going back to states in every other level j, where $0 \leq j \leq i$. Since a node in the GSS can have $H \times (i + 1)$ edges, i^m paths may need to be explored for any reduction of length $m + 1$ that is performed. This results in such algorithms displaying $O(n^{m+1})$ time complexity. Although we will not prove this here, we shall describe a grammar and an example string that illustrate the properties which trigger quartic behaviour in the RNGLR algorithm. Clearly this is sufficient to show that the RNGLR algorithm is not cubic.

Example – Recognition in $O(n^4)$ time

Consider Grammar 6.1, its LR(1) DFA in Figure 6.1 and the associated RN parse table $\mathcal{T}$ in Table 6.1.

$$\begin{aligned} S' &::= S \\ S &::= SSS \mid SS \mid b \end{aligned} \qquad (6.1)$$



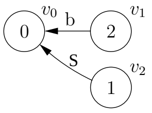
0
$S' ::= \cdot S, \$$
$S ::= \cdot SSS, \{\$, b\}$
$S ::= \cdot SS, \{\$, b\}$
$S ::= \cdot b, \{\$, b\}$
2
$S ::= b\cdot, \{\$, b\}$
1
$S' ::= S\cdot, \$$
$S ::= S \cdot SS, \{\$, b\}$
$S ::= S \cdot S, \{\$, b\}$
$S ::= \cdot SSS, \{\$, b\}$
$S ::= \cdot SS, \{\$, b\}$
$S ::= \cdot b, \{\$, b\}$
3
$S ::= SS \cdot S, \{\$, b\}$
$S ::= SS\cdot, \{\$, b\}$
$S ::= S \cdot SS, \{\$, b\}$
$S ::= S \cdot S, \{\$, b\}$
$S ::= \cdot SSS, \{\$, b\}$
$S ::= \cdot SS, \{\$, b\}$
$S ::= \cdot b, \{\$, b\}$
4
$S ::= SSS\cdot, \{\$, b\}$
$S ::= SS \cdot S, \{\$, b\}$
$S ::= SS\cdot, \{\$, b\}$
$S ::= S \cdot SS, \{\$, b\}$
$S ::= S \cdot S, \{\$, b\}$
$S ::= \cdot SSS, \{\$, b\}$
$S ::= \cdot SS, \{\$, b\}$
$S ::= \cdot b, \{\$, b\}$
b
S
b
b
b
S
S
S

Figure 6.1: The LR(1) DFA for Grammar 6.1.

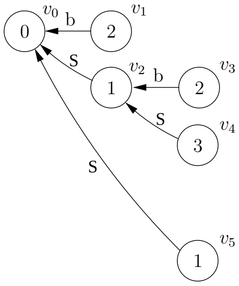
	b	\$	S
0	p2		p1
1	p2	acc	p3
2	r(S,1)	r(S,1)	
3	p2/r(S,2)	r(S,2)	p4
4	p2/r(S,3)/r(S,2)	r(S,3)/r(S,2)	p4

Table 6.1: The RN parse table for Grammar 6.1.

We can illustrate the properties that cause the RNGLR algorithm to have at least $O(n^4)$ time complexity, by parsing the string *bbbbb*. We begin by creating the node v_0 labelled with the start state of the DFA. The shift action $p2$ is the only element in $\mathcal{T}(0, b)$, so we perform the shift and create the new node v_1 labelled 2. As there is the reduce action $r(S, 1)$ in $\mathcal{T}(2, b)$, $(v_0, S, 1)$ is added to the set $\mathcal{R}$. When $(v_0, S, 1)$ is removed from $\mathcal{R}$, the node v_2 labelled 1 is created.

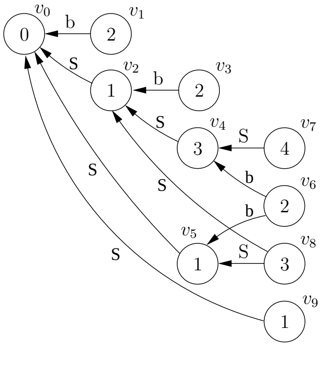


Processing v_2 we find that the shift $p2$ is the only action in $\mathcal{T}(0, b)$ so we create the node v_3 labelled 2 in the next level and add $(v_2, 1)$ since $r(S, 1)$ is in $\mathcal{T}(2, b)$. By performing the reduce in $\mathcal{R}$ we create node v_4 labelled 3 and the edge from v_4 to v_2. Examining the entry in $\mathcal{T}(3, b)$ we find that there is a shift/reduce conflict with actions $p2/r(S, 2)$. We add the actions to the sets $\mathcal{Q}$ and $\mathcal{R}$ respectively. Performing the reduction in $\mathcal{R}$ we create v_5 with an edge to v_0 and then add $(v_5, S, 2)$ to $\mathcal{Q}$.

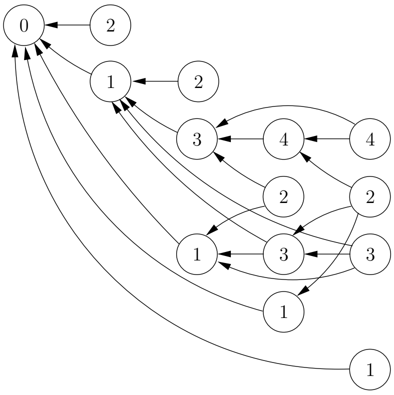


Once the shifts have been removed and processed from the set $\mathcal{Q}$, the new node v_6 labelled 2 has been created in level 3, and $(v_4, S, 1)$ and $(v_5, S, 1)$ are added to $\mathcal{R}$ because of the reduce action in $\mathcal{T}(2, b)$. When both the reductions are done the nodes v_7 and v_8, labelled 4 and 3 respectively, are created along with their associated edges from v_7 to v_4 and v_8 to v_5. At this point $\mathcal{R} = \{(v_4, S, 3), (v_4, S, 2), (v_5, S, 2)\}$ and $\mathcal{Q} = \{(v_7, 2), (v_8, 2)\}$. Removing $(v_4, S, 3)$ from $\mathcal{R}$, the REDUCER traces back a path to v_0 and creates the new node v_9 labelled 1, with an edge between v_9 and v_0. As $\mathcal{T}(1, b)$ contains the shift $p2$, $(v_9, 2)$ is also added to the set $\mathcal{Q}$. When $(v_4, S, 2)$ is processed the REDUCER traces back a path to v_2 and creates a new edge between v_8 and v_2. As a result of the new edge, $(v_2, S, 2)$ is added to $\mathcal{R}$.

When $(v_5, S, 2)$ and then $(v_2, S, 2)$ are processed, the REDUCER traces back a path to v_0 once again, but since the node labelled 1 already exists in the current level and there is an edge to v_0, nothing is done.



After processing the shift actions queued in $\mathcal{Q}$ the node labelled 2 is created in level 4, with edges going back to the nodes labelled 4, 3 and 1 in the previous level.



Notice that the node labelled 4 in the final level of the GSS in Figure 6.2 has edges that go back to every node labelled 4 in the previous levels. This pattern is repeated when recognising any strings of the form b^n. At each level $i \geq 3$ in the GSS there is a node labelled 4 which has edges back to the nodes labelled 3 in each of the levels below i in the GSS. It is this property that is used in [SJE03] to prove that the RNGLR algorithm takes at least $O(n^4)$ time when used to parse sentences b^n using the RN parse table for Grammar 6.1. Since Tomita's Algorithm 1e builds the same GSS as the RNGLR algorithm, they share the property that triggers this worst case behaviour.

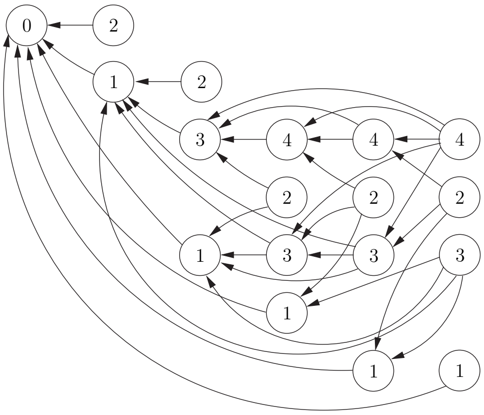


Figure 6.2: The final GSS for Grammar 6.1 and input *bbbbb*.

In fact the RNGLR algorithm is at most $O(n^{M+1})$, where M is the longest rule in the underlying grammar for any RN parse table when $M \geq 3$. Furthermore, grammars in the form of Grammar 6.2 trigger $O(n^M)$ behaviour in such GLR parsers.

$$\begin{aligned} S' &::= S \\ S &::= S^{M-1} \mid b \end{aligned} \tag{6.2}$$

6.2 Achieving cubic time complexity by factoring the grammar

The previous section discussed the properties of grammars that cause GLR recognisers to display polynomial behaviour. Since it is the searching that is done to find the target nodes of a reduction that determines the complexity of such algorithms, an obvious approach to reduce the complexity is to first reduce the searching. The length of the searches done for a reduction is directly related to the length of the grammar's rules. Clearly, by restricting the length of the grammar's rules, an improvement should be possible.

There are several existing algorithms that can transform any context-free grammar into another grammar whose rules have a maximum length of two [HU79]. One of the best known techniques, used by other parsing algorithms such as CYK, is to transform the grammar into Chomsky Normal Form (CNF). Although the algorithm that transforms a grammar to CNF produces a grammar with the desired property, it has two major drawbacks; the resulting parses are done with respect to the CNF grammar and the process to recover the derivations with respect to the original grammar can be expensive, and there is a linear increase in the size of the grammar [HMU01].

Of course, it is not necessary to have CNF to have $O(n^3)$ complexity. All that is required is that the grammar rules all have length at most two. We can achieve this with a grammar which is close to the original by simply factoring the rules. In this way the generated derivations can be closely related to the original grammar.

Grammar 6.4 is the result of factoring Grammar 6.3 so that no right hand side has a length greater than two. Using factored grammars the RNGLR algorithm can parse in at most cubic time.

$$\begin{aligned}
&S' ::= S \\
&S ::= abcB \mid abcD \\
&B ::= d \\
&D ::= d
\end{aligned} \tag{6.3}$$

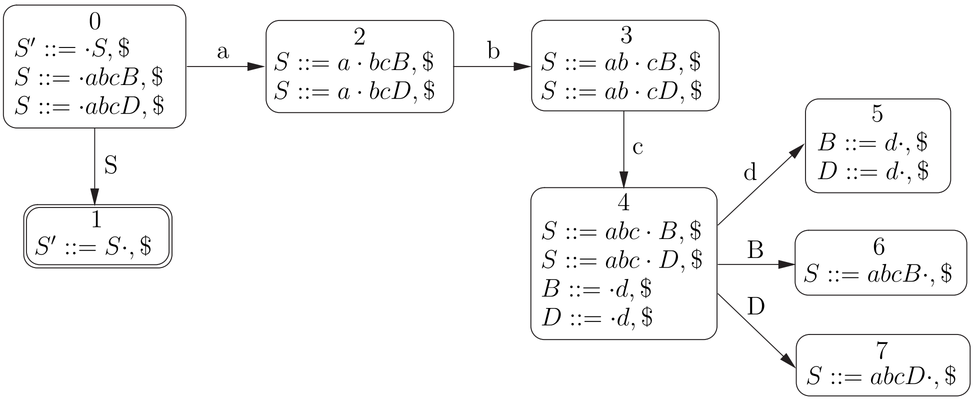


Figure 6.3: The LR(1) DFA for Grammar 6.3.

	a	b	c	d	$	B	D	S
0	p2							p1
1					acc			
2		p3						
3			p4					
4				p5		p6	p7	
5					r(B,1)/r(D,1)			
6					r(S,4)			
7					r(S,4)			

Table 6.2: The RN parse table for Grammar 6.3.

$$\begin{aligned} S' &::= S \\ S &::= aX \\ X &::= bY \\ Y &::= cB \mid cD \\ B &::= d \\ D &::= d \end{aligned} \tag{6.4}$$

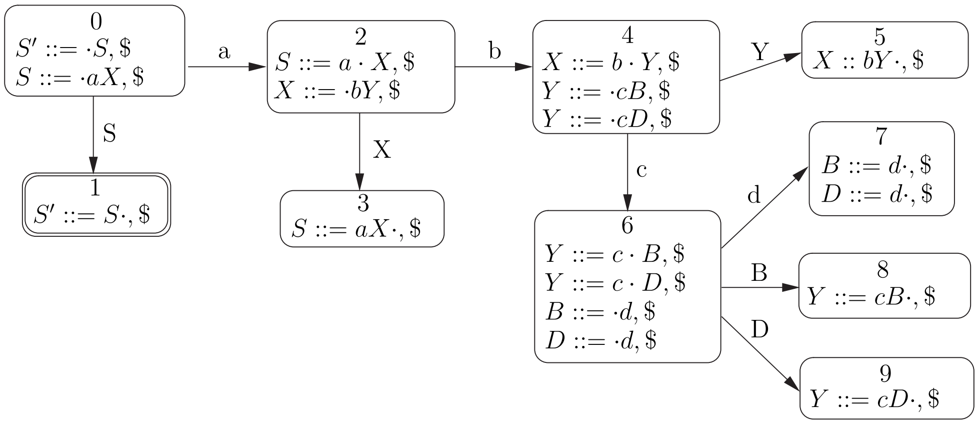


Figure 6.4: The LR(1) DFA for Grammar 6.4.

	a	b	c	d	$	B	D	X	Y	S
0	p2									p1
1					acc					
2		p4						p3		
3					r(S,2)					
4			p6						p5	
5					r(X,2)					
6				p7		p8	p9			
7					r(B,1)/r(D,1)					
8					r(Y,2)					
9					r(Y,2)					

Table 6.3: The RN parse table for Grammar 6.4.

We have to be careful when introducing extra non-terminals. If we only use one non-terminal for the two alternates of S in the following grammar the strings *aba* and *cbc* are incorrectly introduced to the language.

$$\begin{array}{lcl} S' ::= S & & S' ::= S \\ S ::= abc \mid cba & \Rightarrow & S ::= aX \mid cX \\ & & X ::= bc \mid ba \end{array}$$

Thus new non-terminals need to be introduced for each reduction of each rule. Unfortunately this approach can also lead to a substantial increase in the size of the parse table. For example the SLR(1) parse table for our IBM-VS Cobol grammar is 2.8×10^6 cells compared with 6.2×10^6 for the binarised grammar. The increase is even more dramatic for some LR(1) tables. Our ANSI-C grammar has 2.9×10^5

cells compared to the 8.0×10^5 cells of the factorised grammar. For a more detailed discussion of parse tables sizes see Chapter 10.

6.3 Achieving cubic time complexity by modifying the parse table

In the previous section we showed how to use the RNGLR algorithm to achieve cubic worst case complexity by factoring the grammar before parsing. Unfortunately, this technique can dramatically increase the size of the parse table. The objective of factoring the grammar was to restrict the length of the reductions performed during parsing. In this section we present a different approach which achieves the same complexity, but does not increase the size of the parse table to the same degree.

Instead of factoring the grammar it is possible to restrict the length of reductions performed by directly modifying the parse table. This involves the creation of N_A additional states for each non-terminal A, where $(N_A + 2)$ is the longest alternate of A. So if $N_A \geq 1$ then the additional states $A_1 \ldots A_{N_A}$ are created.

In addition to this, a new type of reduction action is added to the parse table so that only reductions with a maximum length of two are performed. The new reductions are of the form $r(A_j, 2)$, where A_j is an additional state and 2 is the length of the reduction. When such an action is performed two symbols are popped off the stack and A_j is pushed onto the stack. For example consider Grammar 6.3 and the associated BRN parse table 6.4. By using the parse table shown in Table 6.4 to parse the string *abcd*, the GSS in Figure 6.5 is constructed.

	a	b	c	d	$	B	D	S
0	p2							p1
1					acc			
2		p3						
3			p4					
4				p5		p6	p7	
5					r(B,1)/r(D,1)			
6					r(8,2)			
7					r(8,2)			
8					r(9,2)			
9					r(S,2)			

Table 6.4: The BRN parse table for Grammar 6.3.

Although the GSS created with the use of the BRN parse table is larger than the one created with the RN table, the increase in size is only a constant factor. Both

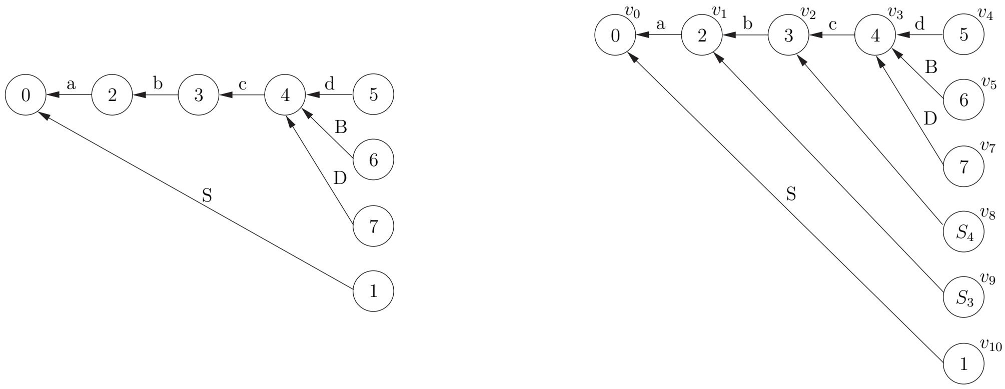


Figure 6.5: The GSS's constructed using parse table 6.2 (left) and parse table 6.4 (right) for the input *abcd*.

GSS's have $O(n^2)$ edges, but the GSS generated by the BRN table can be constructed in at most cubic time.

In [SJE03] a proof is given that shows the BRN parse table accepts exactly the same set of strings as the RN parse table. Since there is another proof in [SJ00] that shows the RN parse table to accept precisely the same strings as an LR(1) parser for the same grammar we can be rely on the parse table being correct.

6.4 The BRNGLR recognition algorithm

This section presents the formal definition of the BRNGLR recognition algorithm that uses BRN parse tables to parse sentences in at most cubic time.

BRNGLR recogniser

```
input data start state $S_S$, accept state $S_A$, RN table $\mathcal{T}$, input string $a_1...a_n$

function PARSER
    set $result = failure$
    if $n = 0$ then
        if $acc \in \mathcal{T}(S_S, \$)$ then set $result = success$
    else
        create a node $v_0$ labelled $S_S$
        set $U_0 = \{v_0\}, \mathcal{R} = \emptyset, \mathcal{Q} = \emptyset, a_{n+1} = \$, U_1 = \emptyset, ..., U_n = \emptyset$
        if $pk \in \mathcal{T}(S_S, a_1)$ then add $(v_0, k)$ to $\mathcal{Q}$
        for all $r(X, 0) \in \mathcal{T}(S_S, a_1)$ do add $(v_0, X, 0)$ to $\mathcal{R}$
        for $i = 0$ to $n$ do
            while $U_i \neq \emptyset$ do
                while $\mathcal{R} \neq \emptyset$ do REDUCER($i$)
            SHIFTER($i$)
        if $S_A \in U_n$ then set $result = success$
    return $result$

function REDUCER($i$)
    remove $(v, X, m)$ from $\mathcal{R}$
    if $m = 2$ then let $\chi$ be the set of children of $v$
    else let $\chi = \{v\}$
    for each node $u \in \chi$ do
        if $X$ is a non-terminal then let $k$ be the label of $u$ and let $pl \in \mathcal{T}(k, X)$
        else let $l = X$
        if there is a node $w \in U_i$ with label $l$ then
            if there is not an edge from $w$ to $u$ then
                create an edge from $w$ to $u$
                if $m \neq 0$ then
                    for all $r(B, t) \in \mathcal{T}(l, a_{i+1})$ where $t \neq 0$ do add $(u, B, t)$ to $\mathcal{R}$
        else
            create a node $w \in U_i$ labelled $l$ and an edge from $w$ to $u$
            if $ph \in \mathcal{T}(l, a_{i+1})$ then add $(w, h)$ to $\mathcal{Q}$
            for all $r(B, 0) \in \mathcal{T}(l, a_{i+1})$ do add $(w, B, 0)$ to $\mathcal{R}$
            if $m \neq 0$ then
                for all $r(B, t) \in \mathcal{T}(l, a_{i+1})$ where $t \neq 0$ do add $(u, B, t)$ to $\mathcal{R}$

function SHIFTER($i$)
    if $i \neq n$ then
        set $\mathcal{Q}' = \emptyset$
        while $\mathcal{Q} \neq \emptyset$ do
            remove an element $(v, k)$ from $\mathcal{Q}$
            if there is a node $w \in U_{i+1}$ with label $k$ then
                create an edge from $w$ to $v$
                for all $r(B, t) \in \mathcal{T}(k, a_{i+2})$ where $t \neq 0$ do add $(v, B, t)$ to $\mathcal{R}$
            else
                create a node $w \in U_{i+1}$ labelled $k$ and an edge from $w$ to $v$
                if $ph \in \mathcal{T}(l, a_{i+2})$ then add $(w, h)$ to $\mathcal{Q}'$
                for all $r(B, 0) \in \mathcal{T}(l, a_{i+2})$ do add $(w, B, 0)$ to $\mathcal{R}$
                for all $r(B, t) \in \mathcal{T}(l, a_{i+2})$ where $t \neq 0$ do add $(v, B, t)$ to $\mathcal{R}$
        copy $\mathcal{Q}'$ into $\mathcal{Q}$
```

Although the above algorithm succeeds in parsing all context-free grammars in at most cubic time, it is disappointing that the parse table increases in size (even though the increase is only by a constant factor). It turns out that because of the regular way the additional reductions are done, it is possible to achieve the same complexity without modifying the parse table. The following section describes such an algorithm.

6.5 Performing 'on-the-fly' reduction path factorisation

The algorithm defined in the previous section is an extension of the RNGLR algorithm which uses BRN parse tables to achieve cubic worst case time complexity. A further extension of the previous algorithm performs this binary translation 'on-the-fly'. This algorithm works on an original RN parse table, but includes the extra machinery necessary to only carry out searches with a maximum length of two. When an action $r(X, m)$ is encountered in the parse table, the algorithm stores the pending reduction in the set $\mathcal{R}$ in the form (v, X, m), where v is the target of the edge down which the reduction is to be applied and m is the length of the reduction. When $m > 2$ a new edge is added from a special *bookkeeping* node labelled X_m in the current level to every child node u of v and the element $(u, X, m-1)$ is added to $\mathcal{R}$. This technique ensures that reductions of length greater than 2 are done in $m-1$ steps of length two. The bookkeeping nodes prevent the repeated traversal of a path in the same way that the extra states added to the modified RN parse table do.

The on-the-fly algorithm is the preferred implementation of the BRNGLR algorithm as it does not increase the size of the RN parse table to achieve cubic worst case time complexity.

BRNGLR 'on-the-fly' recogniser

```
input data start state $S_S$, accept state $S_A$, RN table $\mathcal{T}$, input string $a_1...a_n$

function PARSER
    set $result = failure$
    if $n = 0$ then
        if $acc \in \mathcal{T}(S_S, \$)$ then set $result = success$
    else
        create a node $v_0$ labelled $S_S$
        set $U_0 = \{v_0\}, \mathcal{R} = \emptyset, \mathcal{Q} = \emptyset, a_{n+1} = \$, U_1 = \emptyset, ..., U_n = \emptyset$
        if $pk \in \mathcal{T}(S_S, a_1)$ then add $(v_0, k)$ to $\mathcal{Q}$
        for all $r(X, 0) \in \mathcal{T}(S_S, a_1)$ do add $(v_0, X, 0)$ to $\mathcal{R}$
        for $i = 0$ to $n$ do
            while $U_i \neq \emptyset$ do
                while $\mathcal{R} \neq \emptyset$ do REDUCER($i$)
                Shifter($i$)
        if $S_A \in U_n$ then set $result = success$
    return result

function Reducer($i$)
    remove $(v, X, m)$ from $\mathcal{R}$
    if $m \geq 2$ then let $\chi$ be the set of children of $v$
    else let $\chi = \{v\}$
    if $m \leq 2$ then
        for each node $u \in \chi$ do
            let $k$ be the label of $u$ and let $pl \in \mathcal{T}(k, X)$
            if there is a node $w \in U_i$ with label $l$ then
                if there is not an edge from $w$ to $u$ then
                    create an edge from $w$ to $u$
                    if $m \neq 0$ then
                        for all $r(B, t) \in \mathcal{T}(l, a_{i+1})$ where $t \neq 0$ do add $(u, B, t)$ to $\mathcal{R}$
            else
                create a node $w \in U_i$ labelled $l$ and an edge from $w$ to $u$
                if $ph \in \mathcal{T}(l, a_{i+1})$ then add $(w, h)$ to $\mathcal{Q}$
                for all $r(B, 0) \in \mathcal{T}(l, a_{i+1})$ do add $(w, B, 0)$ to $\mathcal{R}$
                if $m \neq 0$ then
                    for all $r(B, t) \in \mathcal{T}(l, a_{i+1})$ where $t \neq 0$ do add $(u, B, t)$ to $\mathcal{R}$
    else
        if there is not a node $w \in U_i$ with label $X_m$ then create one
        for each node $u \in \chi$ do
            if there is not an edge from $w$ to $u$ then
                create an edge from $w$ to $u$
                add $(u, X, m-1)$ to $\mathcal{R}$

function Shifter($i$)
    if $i \neq n$ then
        set $\mathcal{Q}' = \emptyset$
        while $\mathcal{Q} \neq \emptyset$ do
            remove an element $(v, k)$ from $\mathcal{Q}$
            if there is a node $w \in U_{i+1}$ with label $k$ then
                create an edge from $w$ to $v$
                for all $r(B, t) \in \mathcal{T}(k, a_{i+2})$ where $t \neq 0$ do add $(v, B, t)$ to $\mathcal{R}$
            else
                create a node $w \in U_{i+1}$ labelled $k$ and an edge from $w$ to $v$
                if $ph \in \mathcal{T}(l, a_{i+2})$ then add $(w, h)$ to $\mathcal{Q}'$
        for all $r(B,0) \in \mathcal{T}(l, a_{i+2})$ do add $(w, B, 0)$ to $\mathcal{R}$
        for all $r(B,t) \in \mathcal{T}(l, a_{i+2})$ where $t \neq 0$ do add $(v, B, t)$ to $\mathcal{R}$
    copy $\mathcal{Q}'$ into $\mathcal{Q}$
```

For the formal proofs on the correctness and the complexity of the BRNGLR algorithm see [SJE03].

Example – 'on-the-fly' recognition in $O(n^3)$ time

To demonstrate the operation of the 'on-the-fly' BRNGLR recognition algorithm we shall trace the construction of the GSS for Grammar 6.5 and the input *abcd*.

$$\begin{aligned} S' &::= S \\ S &::= abcd \mid abcD \\ D &::= d \end{aligned} \tag{6.5}$$

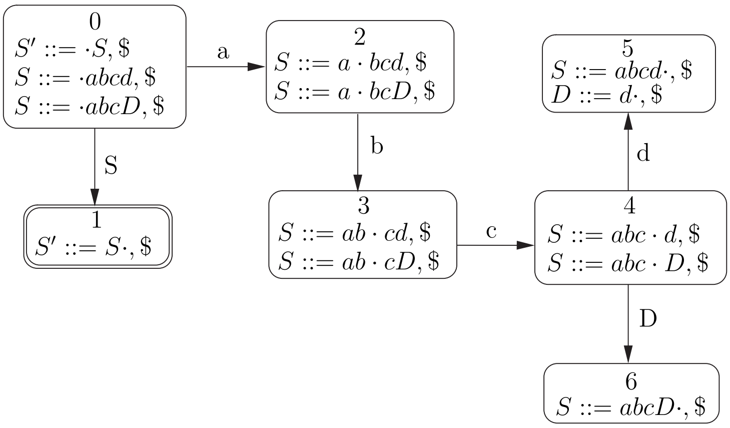
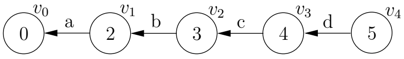


Figure 6.6: The LR(1) DFA for Grammar 6.5.

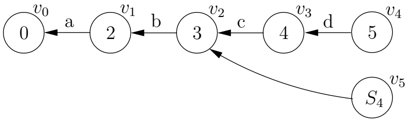
	a	b	c	d	$	D	S
0	p2						p1
1					acc		
2		p3					
3			p4				
4				p5		p6	
5					r(S,4)/r(D,1)		
6					r(S,4)		

Table 6.5: The RN parse table for Grammar 6.5.

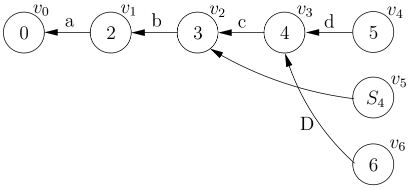
As usual we begin by first creating the node v_0 labelled with the start state of the DFA in Figure 6.6 and then proceed to add the actions found in $\mathcal{T}(0, a)$ to the appropriate sets. In this case, only the shift action $p2$ is added to the set $\mathcal{Q}$, which results in the node v_1 labelled 2 and the edge from v_1 to v_0 being created. Continuing in this way, the next three states created for the remainder of the input string result in the partial GSS shown below.



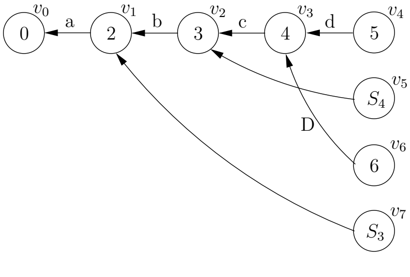
At this point we have $\mathcal{R} = \{(v_3, S, 4), (v_3, D, 1)\}$ and $\mathcal{Q} = \{\}$. When we process $(v_3, S, 4)$ we find v_2, the only child of v_3, and add it to the set χ. Since there is no bookkeeping node labelled S_4 in the current level of the GSS we create one, v_5, add an edge between v_5 and v_3 and then add the additional reduction $(v_2, S, 3)$ to $\mathcal{R}$.



We then process $(v_3, D, 1)$. Since the reduction length is less than 2, we perform a normal reduction which results in the creation of a new node, v_6, labelled 6, with an edge back to v_3. There is a reduction $r(S, 4) \in \mathcal{T}(6, \$)$ so we add $(v_3, S, 4)$ into $\mathcal{R}$.



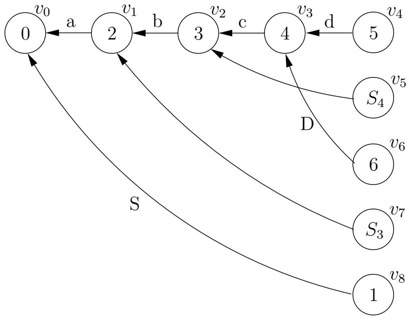
Processing $(v_2, S, 3)$, we find v_1, the only child of v_2, and add it to χ. Since there is no node labelled S_3 in the current level, we create v_7 and add an edge between v_7 and the only node in χ, v_1. The final additional reduction, $(v_1, S, 2)$, is added to $\mathcal{R}$.



When we process the reduction for $(v_3, S, 4)$ we check to see if a bookkeeping node labelled S_4 already exists in the current level. It does. Because we already performed a reduction of length 4 for a rule defined by the non-terminal S, whose path included

v_3, we do not need to continue with the current reduction. Clearly this reduces the amount of edge visits that we perform during the parse.

We continue by performing the reduction for $(v_1, S, 2)$. We find v_0, the only child of v_1 and create the new node v_8 labelled 1, with an edge labelled S to v_0. Since we have consumed all the input and v_8 is labelled by the accept state of the DFA the input *abcd* is successfully accepted. The final GSS is shown below.



6.6 GLR parsing in at most cubic time

We extend the BRNGLR recognition algorithm to a parser by constructing an SPPF in a similar way to the approach we take for the RNGLR algorithm (see Chapter 5). Recall that nodes can be packed if their yields correspond to the same portion of the input string. In order to ensure that a correct SPPF is constructed care needs to be taken when dealing with the bookkeeping SPPF nodes. (Note that the bookkeeping SPPF nodes are not labelled.) We give the algorithm and then illustrate the basic issues using Grammar 6.3. We then discuss the subtleties of the use of bookkeeping nodes in Section 6.7.

BRNGLR parser

```
input data start state $S_S$, accept state $S_A$, RN table $\mathcal{T}$, input string $a_1...a_n$,
  $\epsilon$-SPPF's for each nullable non-terminal and required nullable part $\omega$, and the
  root nodes $u_{I(\omega)}$ of these SPPF's.

function PARSER
  set $result = failure$
  if $n = 0$ then
    if $acc \in \mathcal{T}(0, \$)$ then
      set $result = success$
      output SPPF whose root is $u_{I(S)}$
  else
    create a node $v_0$ labelled $S_S$
```

```
        set $U_0 = \{v_0\}, \mathcal{R} = \emptyset, \mathcal{Q} = \emptyset, a_{n+1} = \$, U_1 = \emptyset, ..., U_n = \emptyset$
        **if** $pk \in \mathcal{T}(S_S, a_1)$ **then** add $(v_0, k)$ to $\mathcal{Q}$
        **for all** $r(X, 0, f) \in \mathcal{T}(0, a_1)$ **do** add $(v_0, X, 0, f, \epsilon)$ to $\mathcal{R}$
        **for** $i = 0$ to $n$ **do**
            **while** $U_i \neq \emptyset$ **do**
                set $\mathcal{N} = \emptyset$
                **while** $\mathcal{R} \neq \emptyset$ **do** Reducer($i$)
                Shifter($i$)
        **if** $S_A \in U_n$ **then**
            set *result* = *success*
            let *root* be the SPPF node that labels the edge $(S_A, v_0)$ in the GSS
            remove nodes in the SPPF not reachable from *root*
            output the SPPF from *root*
         **return** *result*

**function** Reducer($i$)
    remove $(v, X, m, g, y)$ from $\mathcal{R}$
    **if** $m \geq 2$ **then** let $\chi$ be the set of elements $(u, x)$ where $u$ is a child of $v$ and
        the edge $(v, u)$ is labelled $x$
    **else** let $\chi = \{(v, \epsilon)\}$
    **if** $m \leq 2$ **then**
        **for each** node $(u, x) \in \chi$ **do**
            let $k$ be the label of $u$ and let $pl \in \mathcal{T}(k, X)$
            **if** $m = 0$ **then** let $z = u_f$
            **else**
                suppose that $u \in U_j$
                **if** there is no node $z \in \mathcal{N}$ labelled $(X, j)$ **then**
                    create an SPPF node $z$ labelled $(X, j)$
                    add $z$ to $\mathcal{N}$
            **if** there is a node $w \in U_i$ with label $l$ **then**
                /* if the edge exists it will be labelled $z$ */
                **if** there is not an edge from $w$ to $u$ **then**
                    create an edge from $w$ to $u$ labelled $z$
                    **if** $m \neq 0$ **then**
                        **for all** $r(B, t, f) \in \mathcal{T}(l, a_{i+1})$, $t \neq 0$ **do** add $(u, B, t, f, z)$ to $\mathcal{R}$
            **else**
                create a GSS node $w \in U_i$ labelled $l$ and an edge from $w$ to $u$ labelled
                    $z$
                **if** $ph \in \mathcal{T}(l, a_{i+1})$ **then** add $(w, h)$ to $\mathcal{Q}$
                **for all** $r(B, 0, f) \in \mathcal{T}(l, a_{i+1})$ **do** add $(w, B, 0, f, \epsilon)$ to $\mathcal{R}$
                if $m \neq 0$ then
                    for all $r(B,t,f) \in \mathcal{T}(l,a_{i+1})$, $t \neq 0$ do add $(u,B,t,f,z)$ to $\mathcal{R}$
                if $m = 1$ then AddChildren$(z,(y),g)$
                if $m = 2$ then AddChildren$(z,(x,y),g)$
    else
        if there is not a node $w \in U_i$ with label $X_m$ then create one
        for each node $(u,x) \in \chi$ do
            if there is not an edge from $w$ to $u$ then
                create an SPPF intermediate node $z$
                create an edge from $w$ to $u$ labelled $z$
                add $(u,X,m-1,0,z)$ to $\mathcal{R}$
            let $z$ be the label of the edge from $w$ to $u$
            AddChildren$(z,(x,y),g)$

function Shifter$(i)$
    if $i \neq n$ then
        set $\mathcal{Q}^{'} = \emptyset$
        create a new SPPF node $z$ labelled $(a_{i+1},i)$
        while $\mathcal{Q} \neq \emptyset$ do
            remove an element $(v,k)$ from $\mathcal{Q}$
            if there is a node $w \in U_{i+1}$ with label $k$ then
                create an edge from $w$ to $v$ labelled $z$
                for all $r(B,t,f) \in \mathcal{T}(k,a_{i+2})$ where $t \neq 0$ do add $(v,B,t,f,z)$ to $\mathcal{R}$
            else
                create a node $w \in U_{i+1}$ labelled $k$ and an edge from $w$ to $v$ labelled
                    $z$
                if $ph \in \mathcal{T}(k,a_{i+2})$ then add $(w,h)$ to $\mathcal{Q}^{'}$
                for all $r(B,0,f) \in \mathcal{T}(k,a_{i+2})$ do add $(w,B,0,f,\epsilon)$ to $\mathcal{R}$
                for all $r(B,t,f) \in \mathcal{T}(k,a_{i+2})$, $t \neq 0$ do add $(v,B,t,f,z)$ to $\mathcal{R}$
        copy $\mathcal{Q}^{'}$ into $\mathcal{Q}$

function AddChildren$(z,\Delta,f)$
    if $f \neq 0$ then let $\Upsilon = (\Delta,u_f)$
    else let $\Upsilon = \Delta$
    if $z$ does not already have a sequence of children labelled $\Upsilon$ then
        if $z$ has no children then add edges from $z$ to each node in $\Upsilon$
        else
            if $z$ does not have a child which is a packing node then
                create a new packing node $p$ and a tree edge from $z$ to $p$
                add tree edges from $p$ to all the other children of $z$
                remove all tree edges from $z$ apart from the one to $p$
            create a new packing node $t$ and a new tree edge from $z$ to $t$
            create new edges from $t$ to each node in $\Upsilon$
```

Example – parsing in $O(n^3)$ time

To illustrate the operation of the algorithm we shall trace the construction of the GSS and SPPF for the string *abcd* in the language of Grammar 6.3 shown on page 121.

We begin by constructing the GSS node v_0 labelled by the start state of the DFA. Since $p2 \in \mathcal{T}(0, a)$ we create the node v_1 labelled 2, the SPPF node w_0 labelled $(a, 0)$ and the edge from v_1 to v_0 in the GSS which is labelled by w_0.

We proceed to shift the next three input symbols in the same way, which results in the GSS and SPPF shown in Figure 6.7 being constructed.

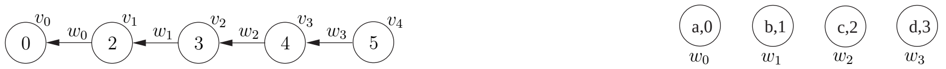


Figure 6.7: The partial GSS and SPPF constructed for Grammar 6.3 and input *abcd* by the BRNGLR 'on-the-fly' algorithm after all the terminals have been shifted.

At this point $\mathcal{R} = \{(v_3, B, 1, 0, w_3), (v_3, D, 1, 0, w_3)\}$ and $\mathcal{Q} = \{\}$. Processing $(v_3, B, 1, 0, w_3)$ results in the creation of node v_5 labelled 6, the SPPF node w_4 and the edge between v_5 and v_3 labelled by w_4. The action in $\mathcal{T}(6, \$)$ is the reduction $r(S, 4)$ so $(v_3, S, 4, 0, w_4)$ is added to $\mathcal{R}$. We then remove $(v_3, D, 1, 0, w_3)$ from $\mathcal{R}$ and process the reduction, which results in the GSS and SPPF shown in Figure 6.8 being constructed and $(v_3, S, 4, 0, w_5)$ added to $\mathcal{R}$.

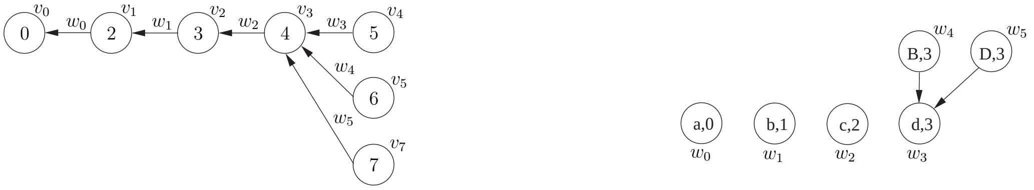


Figure 6.8: The partial GSS and SPPF constructed for Grammar 6.3 and input *abcd* by the BRNGLR 'on-the-fly' algorithm after the two standard reductions are done.

Since the next element, $(v_3, S, 4, 0, w_4)$, that we process from $\mathcal{R}$ has a reduction length greater than two, we collect the children and edges of the reduction's source node, v_3, in the set $\chi = \{(v_2, w_2)\}$. We then create a new bookkeeping node v_8 labelled S_4 and the SPPF node w_6 which labels the new edge from v_8 to v_2. In order to ensure that the correct number of reduction steps are done we also add the binary reduction $(v_2, S, 3, 0, w_6)$ to $\mathcal{R}$. The only operation that remains to be done is to add the SPPF nodes collected in χ and the single node y, which is passed into the REDUCER, as children of the new bookkeeping SPPF node w_6. We do this by calling the ADDCHILDREN function with the parameters $(w_6, (w_2, w_4), 0)$. Since w_6 does not have any existing children, edges to w_2 and w_4 are created. The GSS and SPPF constructed up to this point are shown in Figure 6.9.

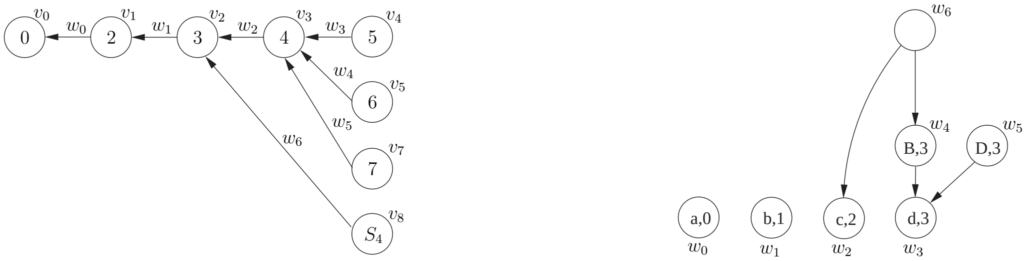


Figure 6.9: The partial GSS and SPPF constructed for Grammar 6.3 and input *abcd* by the BRNGLR 'on-the-fly' algorithm after the first bookkeeping node is created.

When we remove and process the reduction $(v_3, S, 4, 0, w_5)$ from $\mathcal{R}$ we proceed in the same way as before by collecting the children and edges of the reduction's source node in the set $\chi = \{(v_2, w_2)\}$. However, since there already exists a node in the current level that is labelled S_4 we do not create a new one and since there is already an edge from v_8 to v_2 no edge is created either. Instead we make sure that the existing SPPF node w_6 has the correct children by calling ADDCHILDREN with the parameters $(w_6, (w_5, w_2), 0)$.

Since w_6 has a sequence of children that is not labelled (w_5, w_2) we create two new packing nodes as children of w_6 and add the two sequences of SPPF nodes to them as children. This results in the SPPF shown in Figure 6.10 being constructed.

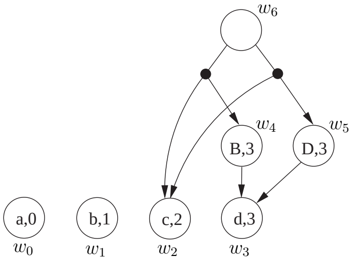


Figure 6.10: The SPPF constructed for Grammar 6.3 and input *abcd* by the BRNGLR 'on-the-fly' algorithm after the packing nodes are created for the bookkeeping node w_6.

At this point $\mathcal{R} = \{(v_2, S, 1, 0, w_6)\}$ and $\mathcal{Q} = \{\}$. Processing $(v_2, S, 3, 0, w_6)$ we add $\{(v_1, w_1)\}$ to χ and since m is greater than 2, we proceed to create a new node v_9 labelled S_3, a new SPPF node w_7 and an edge from v_9 to v_1 which is labelled by w_7. We then add $(v_1, S, 2, 0, w_7)$ for the final reduction step of the binary sequence of reductions to $\mathcal{R}$. Finally, we call ADDCHILDREN$(w_7, (w_6, w_1), 0)$ which adds w_6 and

w_1 as children of w_7 and results in the SPPF shown in Figure 6.11 being constructed.

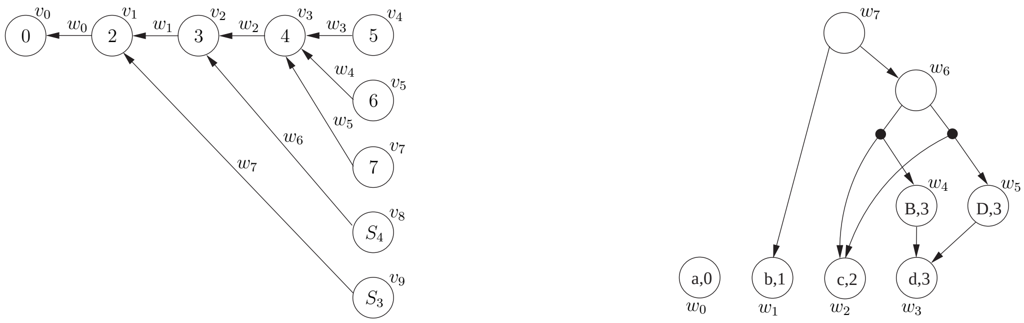


Figure 6.11: The GSS and SPPF constructed for Grammar 6.3 and input *abcd* by the BRNGLR 'on-the-fly' algorithm after the last bookkeeping node is created.

When we process $(v_1, S, 2, 0, w_7)$ we trace back to node v_0 and add (v_0, w_0) to χ. Since $m = 2$ we recognise that this reduction will be completed in this step, so we find the goto action p_1 in $\mathcal{T}(0, S)$. We then create the new node v_{10} labelled 1 and the SPPF node w_8 labelled $(S, 0)$, which we use to label the edge between nodes v_{10} and v_0. The subsequent call to AddChildren with the parameters $(w_8, (w_0, w_7), 0)$ results in the final SPPF shown in Figure 6.12 being constructed.

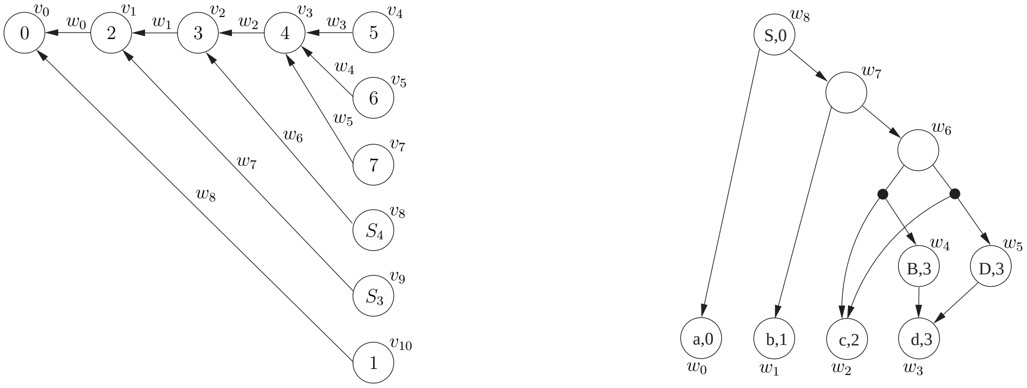


Figure 6.12: Standard and Binary SPPF's for Grammar 6.3 and input *abcd*.

6.7 Packing of bookkeeping SPPF nodes

The previous section demonstrated how the 'on-the-fly' BRNGLR parsing algorithm builds an SPPF during parsing. At first it may appear that in general we could pack some of the bookkeeping nodes in the SPPF more effectively. In fact we cannot do this because incorrect derivations may be introduced to the SPPF. This section highlights the subtlety of packing bookkeeping nodes by tracing the construction of a GSS and SPPF for a grammar and string that will introduce incorrect derivations if the packing is done naïvely.

Example – how to pack bookkeeping SPPF nodes

It is only possible to pack two bookkeeping SPPF nodes when they arise from different reductions whose target is the same state node. Consider Grammar 6.6, the LR(1) DFA in Figure 6.13 and RN parse table in Table 6.6. We can illustrate the consequences of incorrectly packing the bookkeeping SPPF nodes when parsing the string *abc*.

$$\begin{aligned} S' &::= S \\ S &::= abB \mid abD \mid AbB \\ A &::= a \\ B &::= c \\ D &::= c \end{aligned} \tag{6.6}$$

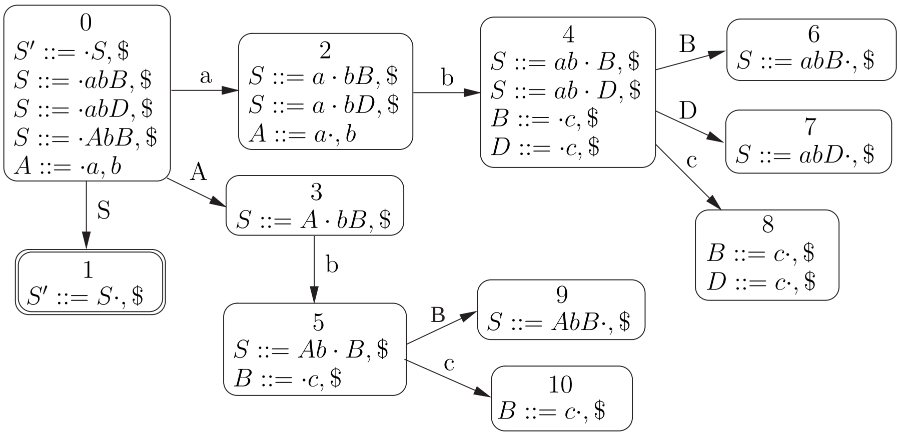


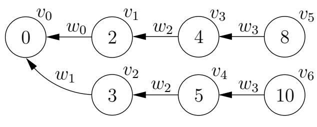
Figure 6.13: The LR(1) DFA for Grammar 6.6.

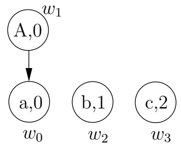
	a	b	c	$	A	B	D	S
0	p2				p3			p1
1				acc				
2		p4/r(A,1,0)						
3		p5						
4			p8			p6	p7	
5			p10			p9		
6					r(S,3,0)			
7					r(S,3,0)			
8					r(B,1,0)/r(D,1,0)			
9					r(S,3,0)			
10					r(B,1,0)			

Table 6.6: The RN parse table for Grammar 6.6.

We begin in the usual way, creating the node v_0 labelled with the start state of the DFA and adding the actions from the associated parse table entry to the sets $\mathcal{R}$ and $\mathcal{Q}$. Since $p2$ is the only action in $\mathcal{T}(0, a)$ we shift the first input symbol and create a new node v_1 labelled 2, an SPPF node w_0 labelled $(a, 0)$ and an edge from v_1 to v_0 labelled by w_0. There is a shift/reduce conflict in the parse table at $\mathcal{T}(2, b)$ so $(v_1, 4)$ and $(v_0, A, 1, 0, w_0)$ are added to the sets $\mathcal{Q}$ and $\mathcal{R}$ respectively. Processing $(v_0, A, 1, 0, w_0)$ results in the node v_1 labelled 3, the SPPF node w_1 labelled $(A, 0)$ and the edge from v_1 to v_0 labelled by w_1 being created. Since the shift action $p5$ is in $\mathcal{T}(3, b)$, $(v_2, 5)$ is added to the set $\mathcal{Q}$. We then call the AddChildren function with the parameters $(v_1, (w_0), 0)$, which results in the SPPF node w_0 being added to w_1 as a child.

At this point, no other reductions are queued in the set $\mathcal{R}$ so the shifts are processed from the set $\mathcal{Q}$ by the Shifter. This results in the new nodes v_3 and v_4 being created, with edges to v_1 and v_2 labelled by the newly created SPPF node w_2. Both nodes v_3 and v_4 only have shift actions in their respective parse table entries so the new nodes v_5 and v_6 are created along with the associated SPPF node w_3.



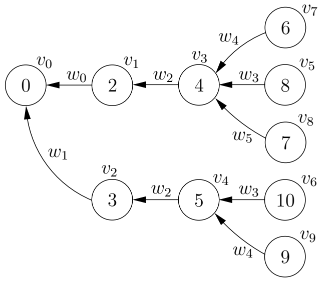


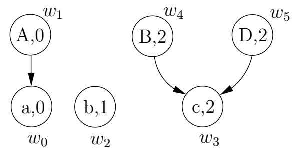
At this point $\mathcal{R} = \{(v_3, B, 1, 0, w_3), (v_3, D, 1, 0, w_3), (v_4, B, 1, 0, w_3)\}$ and $\mathcal{Q} = \{\}$. Processing the first element in $\mathcal{R}$ results in the new node v_7 labelled 6, the SPPF node w_4 labelled $(B, 2)$ and the edge from v_7 to v_3 being created. Since the length of the reduction is one, we find the new reduction $r(S, 3, 0)$ in $\mathcal{T}(6, \$)$ and add $(v_3, S, 3, 0, w_4)$

to $\mathcal{R}$ before proceeding to call AddChildren$(w_4, (w_3), 0)$ which makes w_3 a child of w_4 in the SPPF.

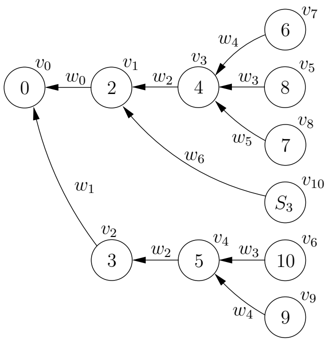
When we remove $(v_3, D, 1, 0, w_3)$ from $\mathcal{R}$ and continue to process the reduction in the Reducer, we create the new node v_8 labelled 7, the SPPF node w_5 labelled $(D, 2)$ and the edge between v_8 and v_3 labelled by w_5. The new reduction $(v_3, S, 3, 0, w_5)$ is added to $\mathcal{R}$ and then AddChildren$(w_5, (w_3), 0)$ is called which results in $w3$ being made a child of w_5.

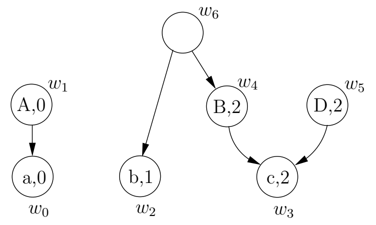
Then we remove $(v_4, B, 1, 0, w_3)$ from $\mathcal{R}$ which results in the node v_9 labelled 9 being created in the GSS. Since an edge does not already exist from v_9 to v_4 one is created. However, because we have already created an SPPF node labelled $(B, 2)$ while constructing the current level, which we find in the set $\mathcal{N}$, we reuse it to label the new edge. In addition to this because the parse table contains the reduction $r(S, 3, 0)$ in $\mathcal{T}(9, \$)$, we add $(v_4, S, 3, 0, w_5)$ to $\mathcal{R}$.



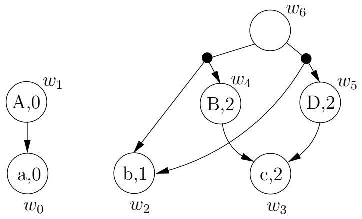


We then continue processing the remaining reductions $(v_3, S, 3, 0, w_4)$, $(v_3, S, 3, 0, w_5)$ and $(v_4, S, 3, 0, w_5)$ from the set $\mathcal{R}$. For $(v_3, S, 3, 0, w_4)$ the Reducer determines that the length of the reduction is greater than two and proceeds to perform a binary reduction. The child node and edge (v_1, w_2), from node v_3 are found and added to the set χ. A new bookkeeping node v_{10} labelled S_3 is then created and an edge is added between v_{10} and v_1. The final binary reduction of the sequence $(v_1, S, 2, 0, w_6)$ is also added to $\mathcal{R}$ before the SPPF is updated by AddChildren$(w_6, (w_2, w_4), 0)$.

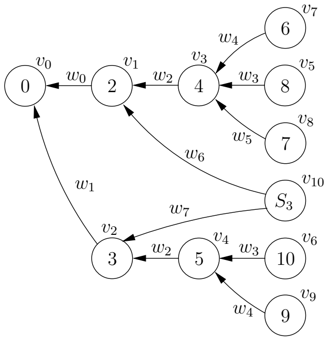


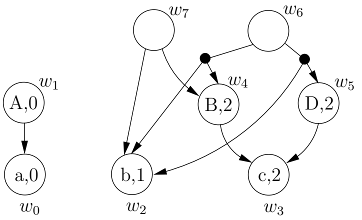


When $(v_3, S, 3, 0, w_5)$ is processed, another binary reduction is performed by the Reducer. Once again the child node and edge (v_1, w_2), from node v_3 are found and added to the set χ. However, since there is already a node in the current level that is labelled S_3, we do not create a new one, but reuse the existing one. In addition to this, because there is already an edge between v_{10} and v_1, we also reuse the edge and SPPF node. When we finally call AddChildren$(w_6, (w_2, w_5), 0)$, we find that two new packing nodes need to be created as children of w_6. So we add the original sequence of children to one packing node and the new sequence that is passed into AddChildren to the other.



When we process $(v_4, S, 3, 0, w_5)$, another binary reduction is performed. We begin by setting $\chi = \{(v_2, w_2)\}$. Since we already have a node labelled S_3 in the current level of the GSS, we can use it again. However, because there is no edge that goes between v_{10} and v_2, we create a new one and label it with a new SPPF node w_7. Before calling the AddChildren function we add the remaining binary reduction $(v_2, S, 2, 0, w_7)$ to $\mathcal{R}$.





Processing the final two reductions, $(v_1, S, 2, 0, w_6)$ and $(v_2, S, 2, 0, w_7)$, held in $\mathcal{R}$ results in the node v_{11} being created in the GSS that is labelled with the accepting state of the DFA. Since v_{11} is in the final level of the GSS, the string *abc* is accepted by the parser. The final GSS and SPPF constructed by the BRNGLR parsing algorithm are shown in Figure 6.14.

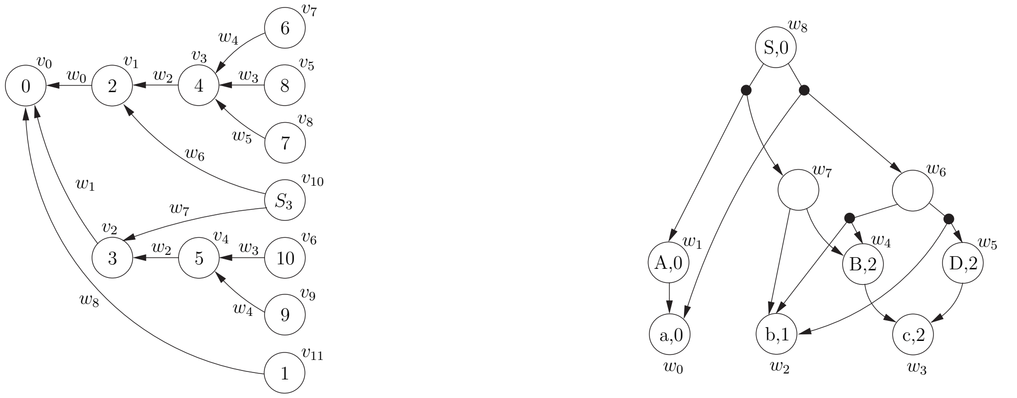


Figure 6.14: The final Binary GSS and SPPF for Grammar 6.6 and input *abc*.

At first, it may appear that it is possible to pack the two SPPF nodes w_6 and w_7 to produce the more compact SPPF shown in Figure 6.15. Unfortunately if this is done the following incorrect 'derivation' will be included.

$$S' \Rightarrow S \Rightarrow AbD \Rightarrow abD \Rightarrow abc$$

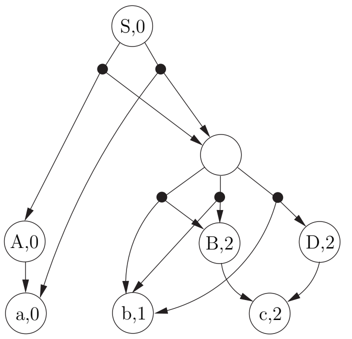


Figure 6.15: An incorrect Binary SPPF for Grammar 6.6 and input *abc* caused by packing too many additional nodes.

6.8 Summary

In this chapter we have presented an algorithm capable of parsing all context-free grammars in at most $O(n^3)$ time and space. Two versions of the recognition algorithm were presented. The first worked on modified RN parse tables which included new states and reduction actions to ensure reductions of at most length two were done. The second algorithm used RN parse tables and split reductions with lengths $m > 2$ into $m - 1$ reductions of length 2 'on-the-fly'. This did not require any modifications to be done to the parse table or the grammar, and hence did not increase the size of the parser.

The 'on-the-fly' recognition algorithm was then extended to a parser which is able to construct an SPPF representation of all possible derivations for a given input string in at most cubic time and space.

Proofs of the correctness of the algorithms and their complexity analysis can be found in [SJE03]. Chapter 10 presents the experimental results for grammars which trigger worst case performance and several programming language grammars and strings. The results corroborate the complexity analysis and provide an encouraging comparison between BRNGLR and other general parsing algorithms.

The next chapter presents another general parsing algorithm that achieves cubic worst case time complexity by minimising the amount of stack activity that is done during parsing.

Chapter 7

Reduction Incorporated Generalised LR parsing

The most efficient general parsing algorithm to date achieves $O(n^{2.376})$ worst case time complexity. Unfortunately, as already discussed in the previous chapter, the high constants of proportionality associated with this technique make it impractical for all but the longest strings.

This thesis focuses on the general parsing algorithms based on Tomita's GLR parsing technique. The initial goal of GLR parsing was to provide an efficient algorithm for "practical natural language grammars" [Tom86] by exploiting the efficiency of Knuth's deterministic LR parser. The BRNGLR algorithm, presented in Chapter 6, is a GLR parsing algorithm that achieves $O(n^3)$ worst case complexity. Although the worst case is not always reached, the run time of the algorithm is far from ideal, especially when compared to the deterministic techniques.

Since the run time of a parser is highly visible to a user, there have been several attempts at speeding the run time of LR parsers [BP98, HW90, Pen86, Pfa90]. Most of these have focused on achieving speed ups by implementing the handle finding automaton in low-level code. A different approach to improving efficiency is presented in [AH99, AHJM01], the basic ethos of which is to reduce the reliance on the stack. It is clear that the run time performance of shift-reduce parsers is dominated by the maintenance of the parse stack. The recognition of regular languages that are defined by regular expressions, is much more efficient than the parsing of context-free languages that are defined by context-free grammars because only the current state needs to be stored. Informally, Aycock and Horspool's idea uses FA based recognition techniques for the regular parts of a grammar and only uses a stack for the parts of the grammar which are not regular. (We shall define this formally in detail below.)

Unfortunately, as the authors point out, the algorithm presented in [AH99] fails to terminate on grammars that contain hidden-left recursion. This chapter presents the Reduction Incorporated Generalised LR (RIGLR) algorithm that is based on the same approach taken by Aycock and Horspool, but which can be used to parse all context-free grammars correctly. As part of the work in this thesis, the RIGLR algorithm was implemented for comparison with the RNGLR and BRNGLR algorithms. The theoretical description of the RIGLR algorithm given here is taken primarily from [SJ03b].

We begin by discussing the role of the stack in a shift-reduce parser and then show how to split a grammar, Γ, into several new grammars for some of the non-terminals that define a regular part of Γ. We then describe the construction of the Intermediate Reduction Incorporated Automata (IRIA) that accept the strings in the language of these regular parts and then use the subset construction algorithm to construct the more deterministic Reduction Incorporated Automata (RIA). Combining the separate RIA to produce the Recursion Call Automaton (RCA) we can recognise the strings in the language of Γ with less stack activity than the GLR parsing techniques. We then introduce the RIGLR algorithm that uses a similar structure to Tomita's GSS to parse all context-free grammars. The chapter concludes with a discussion on the construction of derivation trees for this algorithm.

7.1 The role of the stack in bottom-up parsers

The DFA's associated with the deterministic bottom-up shift-reduce parsers act as handle recognisers for parsing algorithms. A parse is carried out by reading the input string one symbol at a time and performing a traversal through the DFA from its start state. If an accept state is reached from the input consumed, then the leftmost handle of the input has been located. At this point the parser replaces the string of symbols in the input that match the right hand side of the handle's production rule, with the non-terminal on the left hand side of the production rule. Parsing then resumes with the modified input string from the start state of the DFA. If an accept state is reached and the start symbol is the only symbol in the input string, then the original string is accepted by the parser.

The approach of repeatedly feeding the input string into the DFA is clearly inefficient. The initial portion of the input may be read several times before its handle is found. Consequently a stack is used to push symbols on when they are read and pop them off when a handle is found. This prevents the initial portion of the input being repeatedly read.

Many parser generators employ an extended BNF notation (EBNF), which includes regular expression operators to encourage structures to be defined in a way that can be efficiently implemented. Although the stack improves a parser's run time, there are overheads associated with its use, so it is not uncommon for language developers to try and optimise its use; left recursion is preferred to right recursion in bottom-up parsers as the latter causes a deep stack to be created whereas the former yields a shallow stack.

For example, consider the two grammars defined below. Both accept all strings of as, but the one on the left uses left recursion whereas the one on the right uses right recursion.

$$\begin{aligned} S' &::= S \\ S &::= Sa \mid \epsilon \end{aligned} \qquad\qquad \begin{aligned} S' &::= S \\ S &::= aS \mid \epsilon \end{aligned}$$

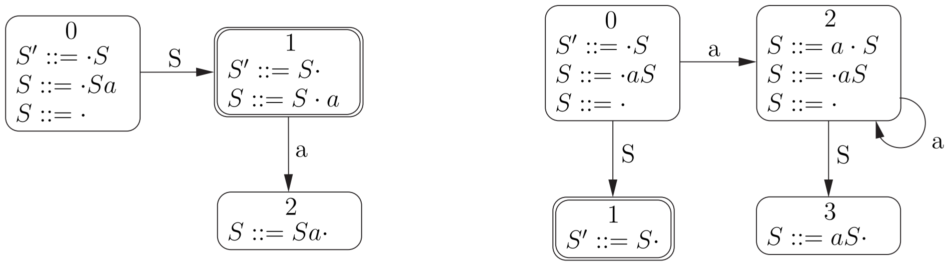


Figure 7.1: The LR(0) DFA's for the left and right recursive grammars given.

A trace of the stacks used during a parse of the string $aaaa$, for both grammars defined above, is shown in Table 7.1. The parse of the right recursive grammar shows that the parser needs to remember the entire left context so it can reduce a the correct number of times. This results in a deeper stack being created.

Left recursion			Right recursion		
Stack	Input	Action	Stack	Input	Action
\$0	a	r2	\$0	a	s2
\$0S1	a	s2	\$0a2	a	s2
\$0S1a2	a	r1	\$0a2a2	a	s2
\$0S1	a	s2	\$0a2a2a2	a	s2
\$0S1a2	a	r1	\$0a2a2a2a2	\$	r2
\$0S1	a	s2	\$0a2a2a2a2S3	\$	r1
\$0S1a2	a	r1	\$0a2a2a2S3	\$	r1
\$0S1	a	s2	\$0a2a2S3	\$	r1
\$0S1a2	\$	r1	\$0a2S3	\$	r1
\$0S1	\$	acc	\$0S1	\$	acc

Table 7.1: Trace of LR(0) parse of input $aaaa$ for the left and right recursive grammars defined above.

Since a grammar's DFA is deterministic, one might expect that a path can simply be re-traced once a handle is found without the use of a stack. If this was the case then we could add an ϵ-edge from each state with a reduction $(X ::= \alpha\cdot)$ to the state at the end of the reduction path whose target is X. For example, consider Grammar 7.1 and the NFA shown in Figure 7.2. We have added reduction transitions, labelled $\mathcal{R}i$ where i is the rule number of the production used in the reduction, to the DFA states that contain items of the form $(X ::= \alpha\cdot)$.

$$\begin{array}{ll} 0. & S' ::= S \\ 1. & S ::= abA \\ 2. & A ::= c \end{array} \tag{7.1}$$

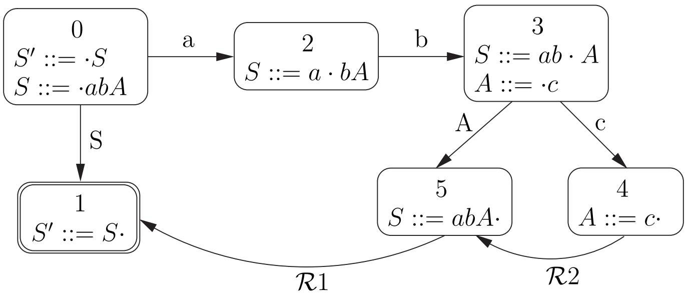


Figure 7.2: The NFA with $\mathcal{R}$-edges for Grammar 7.1.

Recall from the Chomsky hierarchy in Chapter 2 that the FA can be used to recognise languages defined by regular grammars. Although Grammar 7.1 is context-free, it defines a regular language. It is therefore not a big surprise that it can be parsed without the use of a stack. Unfortunately, the problem of deciding whether the language of a context-free grammar is regular or not is known to be undecidable [AU73] and anyway, certain grammars that satisfy this property are difficult to parse without a stack. For example, consider Grammar 7.2 that defines the regular language $\{bad, dac\}$ and the NFA shown in Figure 7.3.

$$\begin{array}{ll} 0. & S' ::= S \\ 1. & S ::= bAd \\ 2. & S ::= dAc \\ 3. & A ::= a \end{array} \tag{7.2}$$

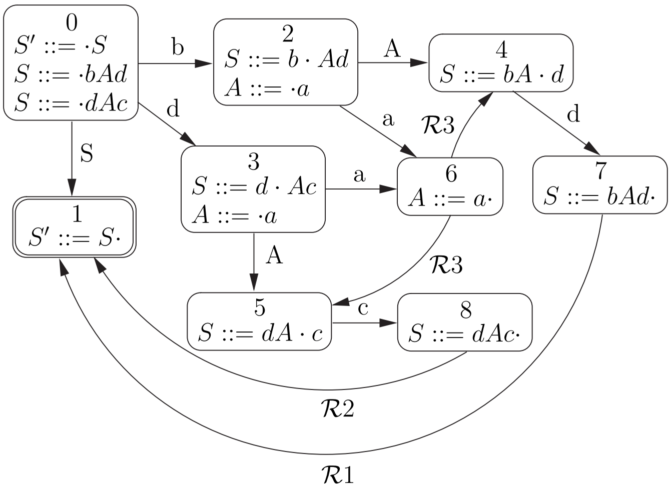


Figure 7.3: The NFA with $\mathcal{R}$-edges for Grammar 7.2.

Using the NFA above it is possible to reach state 6 after consuming *ba* or *da*. The state that we should move to after performing the reduction in state 6 depends upon the path that we have taken to get there. After shifting *ba* we should move to state 4, but after shifting *da* we should move to state 5. However, because there are two $\mathcal{R}$-edges leaving state 6 we do not know which one to follow. Using the NFA in Figure 7.3 it is possible to accept the strings *bad* and *dac* which are not in the language of Grammar 7.2. This is caused by certain multiple instances of non-terminals occurring on the right hand side of production rules. In the example above, it is the non-terminal A that causes the problem. In a standard LR parser, the stack ensures that the correct path is re-traced in such instances preventing incorrect strings being accepted.

Although one may expect that it is possible to recognise some grammars with multiple instances of non-terminals, since they can be regular and by definition accepted by the FA alone, there are some grammars that contain inherently context-free structures. For example, self embedded recursion is a context-free structure that cannot be recognised by a FA alone. Consider Grammar 7.3 and the NFA constructed with the $\mathcal{R}$-edges shown in Figure 7.4.

$$\begin{array}{lll} 0. & S' & ::= S \\ 1. & S & ::= bSd \\ 2. & S & ::= a \end{array} \tag{7.3}$$

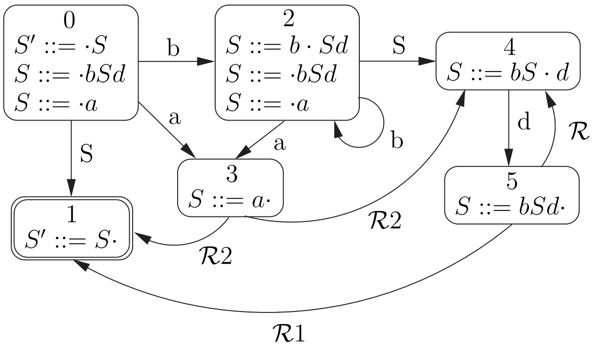


Figure 7.4: The LR(0) DFA with $\mathcal{R}$-edges for Grammar 7.3.

Grammar 7.3 defines the language which contains strings of the form $b^k ad^k$ for any $k \geq 0$. In other words it accepts strings with an equal, or balanced, number of bs and ds. Although the NFA in Figure 7.4 correctly recognises these strings, it also accepts strings of the form $b^i ad^j$ where $i \geq 0$ and $j \geq 0$, which are not in the language defined by the grammar.

The problem with using an NFA without a stack to recognise a context-free language is caused by the NFA not being able to 'remember' what it has already seen. A stack can be used to ensure that once k bs have been shifted, the parser will reduce exactly k times.

7.2 Constructing the Intermediate Reduction Incorporated Automata

We have seen in the previous section that a stack is an important part of any shift-reduce parser. A stack guarantees that:

- when there are multiple instances of non-terminals on the right hand side of the production rules, the parser will move to the correct state after performing a reduction;
- when an instance of self embedded recursion, $A \overset{+}{\Rightarrow} \alpha A \beta$, is encountered during a parse the number of matches to α equals the number of matches to β.

Although a stack is necessary to correctly recognise the portions of a derivation that rely on the self embedded non-terminals, we can deal with the multiple instances of non-terminals by 'multiplying out' some of the states in the FA. Recall, from Chapter 2, that we construct the LR(0) NFA of a grammar by first creating the individual automata for each of the production rules and then join them together with ϵ-edges when a state contains an item with a dot before a non-terminal. For each occurrence of a non-terminal that is encountered in an item, we create a new NFA for that non-terminal's production rule. We now extend this approach by adding extra NFA states for multiple instances of non-terminals. For example, consider Grammar 2.7, on page 31, and the NFA that has been 'multiplied out' in Figure 7.5.

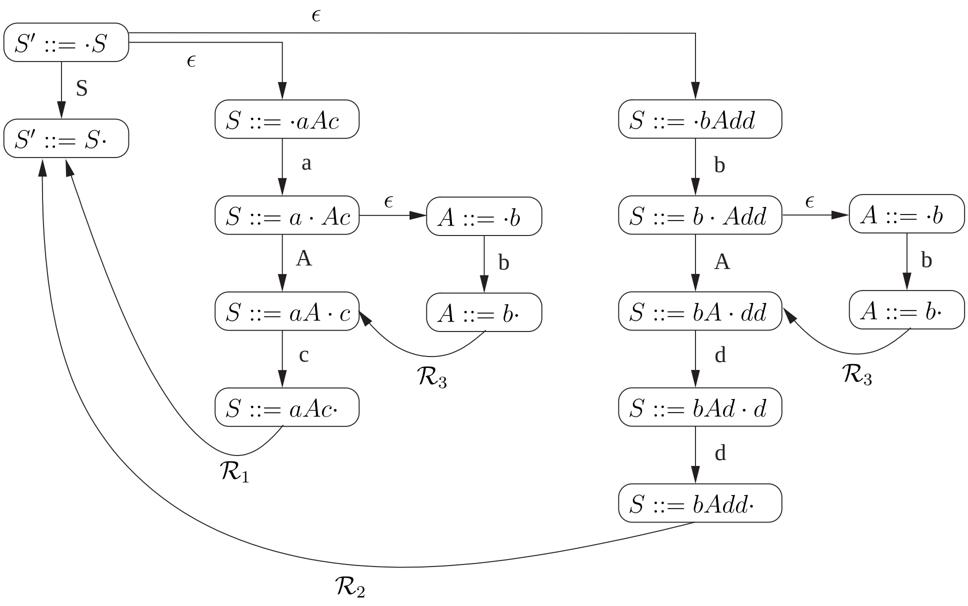


Figure 7.5: The IRIA of Grammar 2.7.

However, if we multiplied out all instances of non-terminals in this way, any recursive rules would result in an infinite number of states being created. For this reason, a recursive instance of a non-terminal B, in a state that contains an item of the form $(A ::= \alpha \cdot B\beta)$, has an ϵ-edge back to the most recent state, on a path from the start state to the current state, that contains an item of the form $(B ::= \cdot\gamma)$. For example consider Grammar 7.4 and the IRIA shown in Figure 7.6.

$$\begin{array}{lll} 0. & S' & ::= S \\ 1. & S & ::= aS \\ 2. & S & ::= bA \\ 3. & A & ::= Ac \\ 4. & A & ::= \epsilon \end{array} \tag{7.4}$$

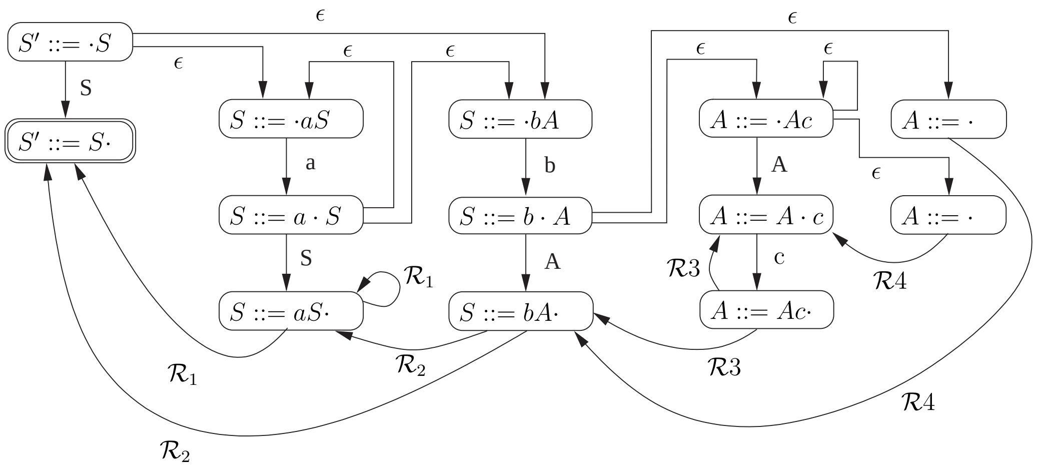


Figure 7.6: The IRIA of Grammar 7.4.

Since the rule $S ::= aS$ is right recursive, the state containing the item $(S ::= a{\cdot}S)$ has ϵ-edges going back to the states labelled by the items $(S ::= \cdot aS)$ and $(S ::= \cdot bA)$. We call the edges which are not created as a result of recursion *primary edges*.

Recall from Chapter 2 that if a grammar contains a non-terminal A such that $A \overset{+}{\Rightarrow} \alpha A \beta$, where $\alpha, \beta \neq \epsilon$, then the grammar contains self-embedding.

A formal definition of the IRIA construction algorithm, taken from [SJ], is given below. It has been proven in [SJ] that for a grammar, Γ, that does not have any self embedded recursion, this algorithm will construct an IRIA that accepts precisely the sentential forms of Γ.

IRIA construction algorithm

Given an augmented grammar Γ (without self embedded recursion) we construct an FA IRIA(Γ) as follows:

Step 1: Create a node labelled $S ::= \cdot S$.

Step 2: While there are nodes in the FA which are not marked as dealt with, carry out the following:

1. Pick a node K labelled $(X ::= \mu \cdot \gamma)$ which is not marked as dealt with.

2. If $\gamma \neq \epsilon$ then let $\gamma = x\gamma'$ where $x \in \mathbf{N} \cup \mathbf{T}$, create a new node, M, labelled $X ::= \mu x \cdot \gamma'$, and add an arrow labelled x from K to M. This arrow is defined to be a *primary edge*.

3. If $x = Y$, where Y is a non-terminal, for each rule $Y ::= \delta$:

   (a) if there is a node L, labelled $Y ::= \cdot\delta$, and a path θ from L to K which consists of only primary edges and primary ϵ-edges (θ may be empty), add an arrow labelled ϵ from K to L. (This new edge is *not* a primary ϵ-edge.)

(b) if (a) does not hold, create a new node with label $Y ::= \cdot\delta$ and add an arrow labelled ϵ from K to this new node. This is defined to be a *primary ϵ-edge.*

4. Mark K as dealt with.

Step 3: Remove all the 'dealt with' marks from all nodes.
Step 4: While there are nodes labelled $Y ::= \gamma\cdot$ that are not dealt with: pick a node K labelled $X ::= x_1 \cdots x_n\cdot$ which is not marked as dealt with. Let $Y ::= \gamma$ be rule i.

If $X \neq S'$ then find each node L labelled $Z ::= \delta \cdot X\rho$ such that there is a path labelled $(\epsilon, x_1, \cdots, x_n)$ from L to K, then add an arrow labelled $\mathcal{R}_i$ from K to the child of L labelled $Z ::= \delta X \cdot \rho$. Mark K as dealt with.

The new edge is called a reduction edge, and if the first (ϵ labelled) edge of the corresponding path is a primary edge then this new edge is defined to be a *primary reduction-edge.*
Step 5: Mark the node labelled $S' ::= \cdot S$ as the start node and mark the node labelled $S' ::= S\cdot$ as the accepting node.

7.3 Reducing non-determinism in the IRIA

It is possible to use an IRIA to guide a parser though a derivation for a given string, but since the automaton is non-deterministic, the parser will encounter a choice of actions in certain states. This section presents the *Reduction Incorporated Automaton* (RIA), a more deterministic automaton than the IRIA which is constructed from the IRIA with the use of the subset construction algorithm.

There are four types of edges in an IRIA. Those labelled by the terminal or non-terminal symbols, the ϵ-edges and the $\mathcal{R}$-edges. Once the $\mathcal{R}$-edges have been created in the IRIA, all the edges labelled with a non-terminal can be removed because they will not be traversed during a parse.

The $\mathcal{R}$-edges are used to locate the target state of a reduction. Since no terminal symbols are consumed during their traversal, it is tempting to treat them as ϵ-edges during the subset construction. However, since many applications are required to produce a derivation of the input after a parse is complete, we cannot simply combine the states that can be reached by $\mathcal{R}$-edges that are labelled with different reductions. Instead we treat them in the same way that we treat the edges labelled by the terminal symbols.

An RIA is constructed from an IRIA by removing the edges labelled by the non-terminal symbols and then performing the subset construction, treating the $\mathcal{R}$-edges as non-empty edges. For example, the RIA in Figure 7.7 was constructed in this way from the IRIA in Figure 7.6.

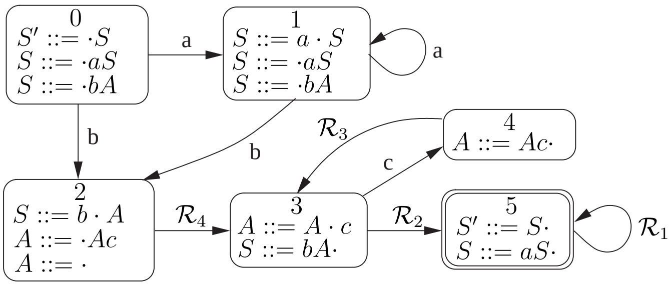


Figure 7.7: The RIA of Grammar 7.4.

7.4 Regular recognition

This section describes how to construct the PDA of a context-free grammar Γ, that recognises strings in the language of Γ with a reduced amount of stack activity compared to GLR recognisers. The approach we take is an extension of the method described by Aycock and Horspool in [AH99]. The description of the algorithm is taken from [SJ03b, JS03, SJ].

7.4.1 Recursion call automata

Recall that a grammar has self embedded recursion if it contains a non-terminal A, such that $A \overset{+}{\Rightarrow} \alpha A \beta$ where both α and $\beta \neq \epsilon$. The structures expressed by self embedded recursion are inherently context-free and hence require a stack to be used to ensure that only valid strings are recognised.

We have already established that the amount of stack activity used during recognition can be reduced if we only use it to recognise these context-free structures. By locating the places that self embedded recursion occurs, we can build an automaton that only uses a stack at these places. This automaton is called the Recursion Call Automaton (RCA).

To construct an RCA, we first need to break any self embedded recursion in the grammar. We can achieve this by effectively 'terminalising' the non-terminals that appear in the self embedded production rules. We replace a non-terminal A in a production rule of the form $X ::= \alpha A \beta$, that has a derivation $A \overset{+}{\Rightarrow} \alpha A \beta$, with the special terminal symbol $A^{\perp}$, in a way that breaks the derivation. We call a grammar Γ that has had all instances of self embedded recursion removed in this way a *derived grammar* of Γ and denote it by Γ_S.

In order to ensure that only the correct derivations of a string are produced (see Section 7.8), we require that any hidden-left recursion is also removed from a grammar before the RCA is constructed. We call the grammar that does not contain any self embedded recursion or hidden-left recursion, the *derived parser grammar* of Γ.

To build the RCA for a derived parser grammar of a context-free grammar Γ, we build a separate RIA for each of the non-terminals defined in Γ and then link them together. For each of the non-terminals A (except S' and S) we create a new rule $S_A ::= A$ in Γ and consider the grammar Γ_A, which has the same rules as Γ but with the new start rule $S_A ::= A$. We then construct the IRIA and RIA for each Γ_A. Once all of the separate, disjoint, RIA have been created, we link them together by removing an edge from a state h to a state k that is labelled $A^{\perp}$ and add a new edge labelled $p(k)$ from h to the start state of the RIA(Γ_A). In addition to this all the accept states of the RIA are labelled with a *pop*. The start state and accepting states of the RCA are the same as the start and accept states of Γ_S. For example consider Grammar 7.5.

$$\begin{array}{lll}
0. & S' & ::= S \\
1. & S & ::= cA \\
2. & A & ::= bAd \\
3. & A & ::= a
\end{array} \tag{7.5}$$

Since the non-terminal A is self embedded, our first step is to terminalise it to produce Grammar 7.6.

$$\begin{array}{lll}
0. & S' & ::= S \\
1. & S & ::= cA \\
2. & A & ::= bA^{\perp}d \\
3. & A & ::= a
\end{array} \tag{7.6}$$

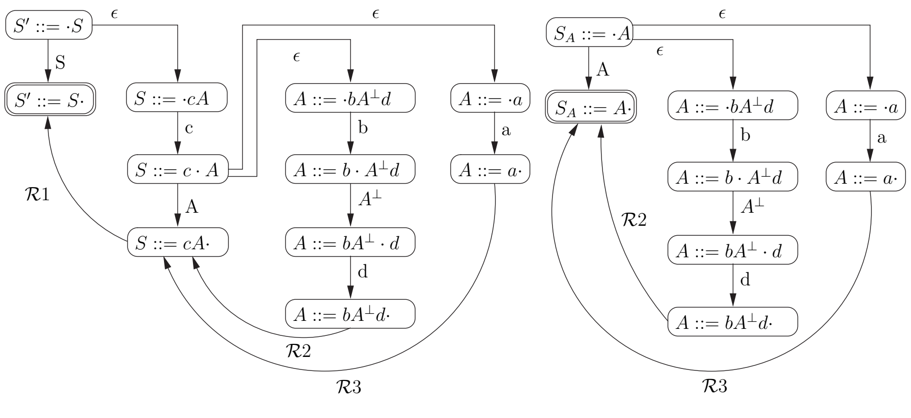


Figure 7.8: The IRIA(Γ_S) and IRIA(Γ_A) of Grammar 7.6.

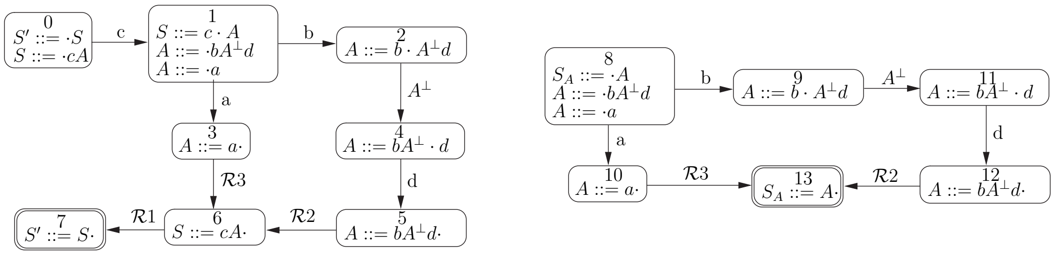


Figure 7.9: The RIA(Γ_S) and RIA(Γ_A) of Grammar 7.6.

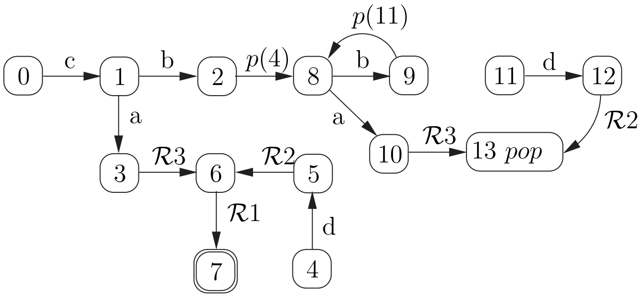


Figure 7.10: The RCA of Grammar 7.6.

It has been proven in [SJ] that a string u is only in the language of a context-free grammar Γ, if the RCA of Γ accepts u.

7.4.2 Parse table representation of RCA

It is often convenient to represent an RCA(Γ) as a parse table, $\mathcal{T}(\Gamma)$, where the rows of the table are labelled by the states of the automaton and the columns by the terminal symbols of Γ and the \$ symbol. The parse table entries contain sets of actions corresponding to the actions associated with the states and edges of the RCA. For all edges from state h to state k, if the edge is labelled by:

- a terminal x, then sk is in $\mathcal{T}(h, x)$;
- $\mathcal{R}i$, then $\mathcal{R}(i, k)$ is in all the columns of row h in $\mathcal{T}$;
- $p(l)$, then $p(l, k)$ is in all the columns of row h in $\mathcal{T}$;

(In this version no lookahead is being employed.)

In addition to the actions above, if a state h in the RCA is labelled *pop*, then every column of state h in $\mathcal{T}$ also contains a *pop* action. So for example, the parse table of the RCA in Figure 7.10 is shown in Table 7.2.

	a	b	c	d	$	A	S
0			s1				
1	s3	s2					
2	p(4,8)	p(4,8)	p(4,8)	p(4,8)	p(4,8)	p(4,8)	p(4,8)
3	$\mathcal{R}(3,6)$	$\mathcal{R}(3,6)$	$\mathcal{R}(3,6)$	$\mathcal{R}(3,6)$	$\mathcal{R}(3,6)$	$\mathcal{R}(3,6)$	$\mathcal{R}(3,6)$
4				s5			
5	$\mathcal{R}(2,6)$	$\mathcal{R}(2,6)$	$\mathcal{R}(2,6)$	$\mathcal{R}(2,6)$	$\mathcal{R}(2,6)$	$\mathcal{R}(2,6)$	$\mathcal{R}(2,6)$
6	$\mathcal{R}(1,7)$	$\mathcal{R}(1,7)$	$\mathcal{R}(1,7)$	$\mathcal{R}(1,7)$	$\mathcal{R}(1,7)$	$\mathcal{R}(1,7)$	$\mathcal{R}(1,7)$
7					acc		
8	s10	s9					
9	p(11,8)	p(11,8)	p(11,8)	p(11,8)	p(11,8)	p(11,8)	p(11,8)
10	$\mathcal{R}(3,13)$	$\mathcal{R}(3,13)$	$\mathcal{R}(3,13)$	$\mathcal{R}(3,13)$	$\mathcal{R}(3,13)$	$\mathcal{R}(3,13)$	$\mathcal{R}(3,13)$
11	$\mathcal{R}(2,13)$	$\mathcal{R}(2,13)$	$\mathcal{R}(2,13)$	$\mathcal{R}(2,13)$	$\mathcal{R}(2,13)$	$\mathcal{R}(2,13)$	$\mathcal{R}(2,13)$
12				s11			
13	pop	pop	pop	pop	pop	pop	pop

Table 7.2: The parse table of the RCA in Figure 7.10.

7.5 Generalised regular recognition

This section introduces the RIGLR recognition algorithm which finds a traversal of an RCA for a string $a_1 \ldots a_n$ if one exists. We begin by providing an informal description of the algorithm and then discuss some specific example grammars that need to be handled with care to ensure that a correct parse is achieved. There is a formal definition of the algorithm at the end of the section.

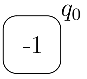
If there is a traversal, for a given string through an automaton, then that string is in the language defined by the automaton. A straightforward way of determining whether a string is in the language defined by an automaton is to traverse the automaton until all the input has been consumed and an accept state is reached. We can take this approach to traverse an RCA, but since it can be non-deterministic, there may be more than one traversal through the automaton that leads to an accept state for a given string. We are interested in finding all such paths, so we employ a breadth first search approach to follow all possible traversals when a choice arises.

A straightforward approach to traversing such a non-deterministic automaton is to maintain a set of states that can be reached by traversing the edges that do not consume any input symbols. In this case, it is the edges labelled by $\mathcal{R}$. We achieve this by maintaining a set U_i, where $0 \leq i \leq n$, during the parse of a string $a_1 \ldots a_n$. We begin by constructing the set U_0 that contains the start state of the RCA and then add, in a similar way to the standard subset construction algorithm [ASU86], all the states that can be reached by traversing the edges from the start state that do not consume any input symbols. When no more states can be added to the set U_0 its construction is complete. We then proceed to create the set U_{i+1} from U_i by adding the states that can be reached by traversing an edge labelled with the current input symbol, a_{i+1}, from each state in U_i. An input string is accepted, if after consuming all input symbols the set U_n contains the RCA's accept state.

This approach only works if the RCA's underlying language does not contain any nested structures. If it does, then the RCA will contain push transitions labelled $p(X)$, where X is a state number. When such an edge is traversed it is necessary to remember X, since a state containing a pop action will eventually be reached that requires the parser to goto X. It is tempting to store the return state in the set U, along with the action's source state, but the possibility of nested pushes and multiple paths caused by the non-determinism would make this approach inefficient.

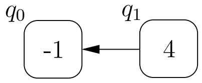
We take the approach described by Aycock and Horspool in [AH99] and use a graph structure, similar to Tomita's GSS (see Chapter 4), to record the return states of the push actions. We call this graph structure the *Recursive Call Graph* (RCG). When we encounter a push action during the traversal of the RCA with a return state l and a target state k, we create a node q labelled l in the RCG and add the pair (k, q) to the current set U_i.

Consider Grammar 7.5 and the RCA constructed in Section 7.4. We parse the string *cbad* by first creating the base node, q_0, of the RCG, labelled -1 (which is not the state number of any RCA state), and then add the element $(0, q_0)$ to the set U_0. Since the only edge that leaves state 0 is labelled by the terminal c, which matches the first symbol on the input string, we move to state 1, and construct the new set $U_1 = \{(1, q_0)\}$.



$U_1 = \{(1, q_0)\}$

There are two edges from state 1, but since they are both labelled by terminal symbols we do not add anything to U_1 in this step. The next input symbol is b, so we traverse the edge to state 2, create $U_2 = \{(2, q_0)\}$. Since there is an edge labelled $p(4)$ from state 2, we create the new RCG node, q_1, labelled 4 and then add $(8, q_1)$ to U_2.



$U_2 = \{(2, q_0), (8, q_1)\}$

The only edges leaving state 8 are labelled by terminal symbols, so we shift on the next input symbol which takes us from state 8, to state 10. We traverse the $\mathcal{R}$-edges so that $U_3 = \{(10, q_1), (13, q_1)\}$. Since state 13 contains a pop action, we pop 4, which is the label of node q_1 in the RCG, and add the element $(4, q_0)$ to U_3.

At this point we have $U_3 = \{(10, q_1), (13, q_1), (4, q_0)\}$ and since there are no more edges that can be traversed without consuming any input symbols the third step of the algorithm is complete.

We then read the final symbol from the input string and traverse the edge labelled d to state 5 creating $U_4 = \{(5, q_0)\}$. We traverse the $\mathcal{R}$-edges and construct $U_4 = \{(5, q_0), (6, q_0), (7, q_0)\}$. Since the next input symbol is the end-of-string symbol, \$, and U contains the element $(7, q_0)$ which has the accept state of the RCA and the base node of the stack, the input *cbad* is accepted.

Before we give the formal definition of the algorithm, we will discuss and show the construction of the RCG for three example grammars that can cause problems if they are not handled with care.

Example – right and hidden-left recursion

Right and hidden-left recursive grammars cause loops of reductions to be created in the RCA. The RIGLR algorithm works by first doing all the reductions that are possible from a state in the RCA before doing any of the other actions. When there are loops of reductions in the RCA, care needs to be taken that the traversal algorithm will terminate. To ensure this we only add the pair (q, k) to the set U_i once in each step of the algorithm.

For example, consider the right recursive Grammar 7.4 on page 149, and the RCA shown in Figure 7.11 that has been constructed from the RIA in Figure 7.7 on page 152.

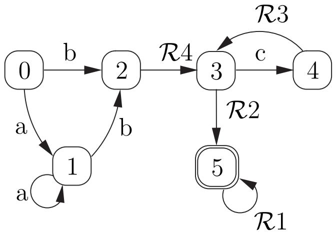


Figure 7.11: The RCA of Grammar 7.4.

To parse the string *ab* we begin by constructing the base node of the RCG labelled -1 and add $(0, q_0)$ to the set U. Since there are no push or pop actions in the RCA, only the base node of the RCG will be used during the parse. As a result we shall not show the RCG during this example.

The first input symbol is a, so we traverse the edge from state 0 to state 1 and add $(1, q_0)$ to U_1. Since the only transitions from state 1 are labelled by terminal symbols, we read the final input symbol, b and traverse the edge to state 2. At this point $U_2 = \{(2, q_0)\}$. We continue by traversing the edge labelled $\mathcal{R}4$ to state 3 and from there the $\mathcal{R}2$ edge to the accepting state 5, adding $(3, q_0)$ and $(5, q_0)$ to U_2.

Although we can accept the input at this point, there is an edge labelled $\mathcal{R}1$ from the accepting state which still needs to be traversed. However, since the edge loops back to the same state without consuming any input symbols, we run the risk of repeatedly adding $(5, q_0)$ to U_2. For this reason, we do not remove elements from U_i once their state has been processed and ensure that no element is added to U_i more than once.

Example – further issues surrounding hidden-left recursion

When a push action is encountered, the traversal algorithm adds a new state to the RCG. If a grammar has a self embedded, hidden-left recursive non-terminal, the traversal algorithm will fail to terminate if a new state is added for every push action. This is because of a loop in the RCA that will not consume any input symbols before doing a push. To prevent the algorithm from failing to terminate, we take a similar approach to Farshi's modification of Tomita's algorithm – we introduce loops in the RCG.

To achieve this we maintain a list, $\mathcal{P}_i$, of the RCG nodes constructed in each step of the algorithm. If a node with the same label has already been constructed we re-use it. $\mathcal{P}_i$ is initialised with the base node and is cleared after an input symbol is read. (To help to see what is going on when we draw an RCG we put nodes constructed at the same step in the algorithm vertically above each other in the RCG.)

So, for example consider the parse of the string bc in the language of Grammar 7.7 [SJ03b]. Notice that the non-terminal S is both self embedded and hidden-left recursive.

$$\begin{array}{lll} 0. & S' & ::= S \\ 1. & S & ::= ASc \\ 2. & S & ::= b \\ 3. & A & ::= a \\ 4. & A & ::= \epsilon \end{array} \tag{7.7}$$

Since the grammar contains self embedded recursion, we terminalise the grammar to produce Grammar 7.8. The IRIA, RIA and RCA constructed for Grammar 7.8 are shown in Figures 7.12, 7.13 and 7.14 respectively.

$$\begin{array}{lll} 0. & S' & ::= S \\ 1. & S & ::= AS^{\perp}c \\ 2. & S & ::= b \\ 3. & A & ::= a \\ 4. & A & ::= \epsilon \end{array} \tag{7.8}$$

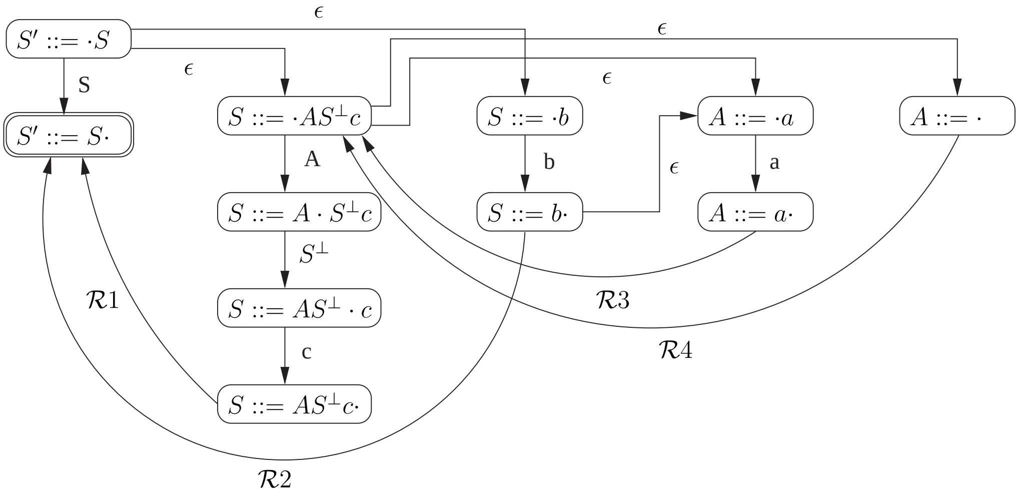


Figure 7.12: The IRIA(Γ_S) of Grammar 7.8.

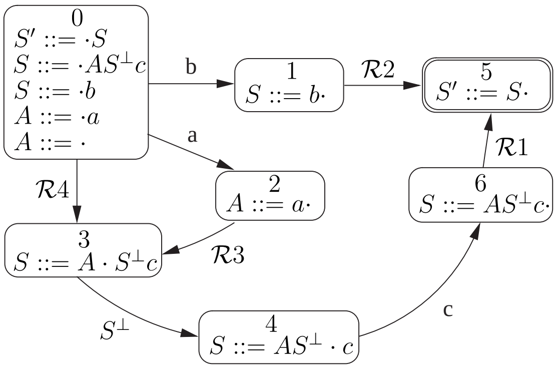


Figure 7.13: The RIA(Γ_S) of Grammar 7.8.

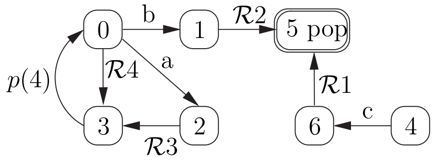
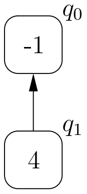


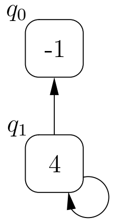
Figure 7.14: The RCA of Grammar 7.8.

We begin the parse by creating the base node of the RCG and adding the element $(0, q_0)$ to the set U_0 and $\mathcal{P}_i$. Before consuming any of the input string it is necessary to traverse any edges labelled by push or $\mathcal{R}$ actions. From state 0 we traverse the edge labelled $\mathcal{R}4$ to state 3, add $(3, q_0)$ to U_0 and then traverse the edge labelled by $p(4)$. The push action results in a new node, q_1, labelled 4, being created in the RCG with an edge back to node q_0. The state of the RCG and the contents of the set U_0 and $\mathcal{P}_i$, at this point of the parse, are shown below.



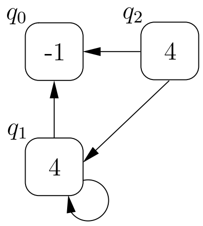
$$\begin{aligned} U_0 &= \{(0,q_0),(3,q_0),(0,q_1)\} \\ \mathcal{P}_0 &= \{(0,q_0),(0,q_1)\} \end{aligned}$$

It is necessary to traverse the reduction transition, $\mathcal{R}4$, from state 0 once again for the process $(0,q_1)$. Performing the reduction we add $(3,q_1)$ to U_0 and then proceed to traverse the push transition $p(4)$. Since there is already a node, q_1, labelled 4 in $\mathcal{P}_0$ we shall re-use it and simply add a loop to it as shown below.



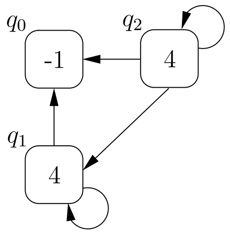
$$\begin{aligned} U_0 &= \{(0,q_0),(3,q_0),(0,q_1),(3,q_1)\} \\ \mathcal{P}_0 &= \{(0,q_0),(0,q_1)\} \end{aligned}$$

There are no more edges that can be traversed from the RCA nodes in U_0 that do not consume any input symbols so the first step of the algorithm is complete. We read the next input symbol, a, and construct $U_1 = \{(2,q_0),(2,q_1)\}$ and set $\mathcal{P}_1 = \emptyset$. We then traverse the edge labelled $\mathcal{R}3$ to state 3 and add $(3,q_0)$ and $(3,q_1)$ to U_1. From state 3 there is the edge labelled $p(4)$ so we create a new RCG node, q_2, labelled 4 with edges back to q_0 and q_1 and add $(0,q_2)$ to U_1 and $\mathcal{P}_1$.



$$\begin{aligned} U_1 &= \{(2,q_0),(2,q_1),(3,q_0),(3,q_1),(0,q_2)\} \\ \mathcal{P}_1 &= \{(0,q_2)\} \end{aligned}$$

From state 0 there is a reduction edge labelled $\mathcal{R}4$ that goes back to state 3. We traverse the edge, add $(3,q_2)$ to U_1 and then traverse the push edge once again. Since node q_2, that is labelled 4, has already been created during this step of the algorithm, we add a loop to it and do not create any new nodes.



$$\begin{aligned} U_1 &= \{(2,q_0),(2,q_1),(3,q_0),(3,q_1),(0,q_2),(3,q_2)\} \\ \mathcal{P}_1 &= \{(0,q_2)\} \end{aligned}$$

We proceed by reading the next input symbol, b, performing the traversal from state 0 to state 1 and then construct $U_2 = \{(1, q_2)\}$ and set $\mathcal{P}_2 = \emptyset$. We then traverse the reduction edge labelled $\mathcal{R}2$ from state 1 to state 5 and add $(5, q_2)$ to U_2. Since state 5 contains a pop action, we find the children, q_0, q_1, q_2, of the RCG node q_2 and add the new elements $(4, q_0), (4, q_1)$ and $(4, q_2)$ to U_2.

That completes step 2 of the algorithm, so we read the final input symbol, c, traverse the edge labelled c from state 4 to state 6 and construct $U_3 = \{(6, q_0), (6, q_1), (6, q_2)\}$ and set $\mathcal{P}_3 = \emptyset$. Traversing the reduction, $\mathcal{R}1$, from state 6 to state 5, we add $(5, q_0), (5, q_1)$ and $(5, q_2)$ to U_3. We then perform the pop actions associated with state 5 of the RCA for $(5, q_1)$ and $(5, q_2)$ which result in $(4, q_0)$ and $(4, q_1)$ respectively, being added to U_3. No pop is done for $(5, q_0)$ since q_0 is the base node of the RCG. At this point $U_3 = \{((6, q_0), (6, q_1), (6, q_2), (5, q_0), (5, q_1), (5, q_2)(4, q_0), (4, q_1)\}$.

Since all the input has been consumed and the process $(5, q_0)$ is in U_3, where state 5 is the accepting state of the RCA and q_0 is the RCG's base node, the input string abc is accepted.

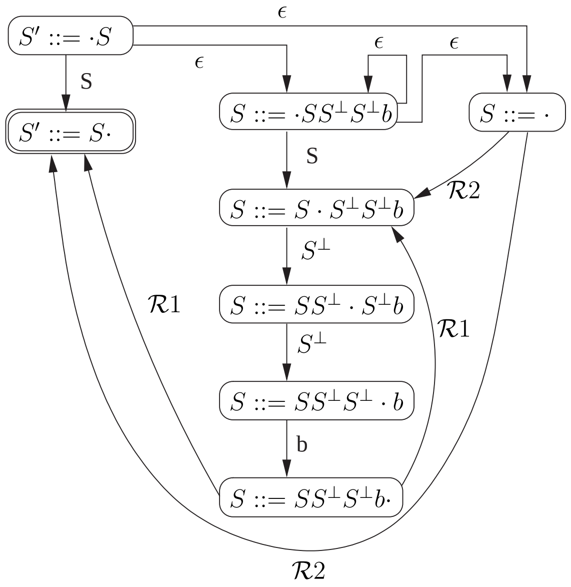
7.5.1 Example – ensuring all pop actions are done

When a pop action is performed by the algorithm on a node q with label h that has an edge to another node p in the RCG, the element (h, p) is added to the set U_i. If a new edge is added from node q to another node w, in the same step of the algorithm, then we need to perform the *pop* action down this new edge. To ensure that this is done, when we add a new edge between q and w, we check to see if U_i contains a process which results in a pop action being performed from q. If such a process exists then we make sure that U_i contains the process (h, w).

For example, consider Grammar 7.9, taken from [SJ03b], and the terminalised version, Grammar 7.10. The associated IRIA, RIA and RCA are shown in Figures 7.15, 7.16 and 7.17 respectively.

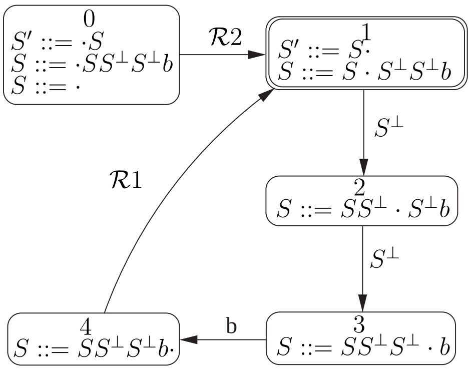
$$\begin{array}{lll} 0. & S' & ::= S \\ 1. & S & ::= SSSb \\ 2. & S & ::= \epsilon \end{array} \tag{7.9}$$

$$\begin{array}{lll} 0. & S' & ::= S \\ 1. & S & ::= SS^{\perp}S^{\perp}b \\ 2. & S & ::= \epsilon \end{array} \tag{7.10}$$



ϵ
$S' ::= \cdot S$
ϵ
ϵ
ϵ
S
$S ::= \cdot SS^{\perp}S^{\perp}b$
$S ::= \cdot$
$S' ::= S\cdot$
S
$\mathcal{R}2$
$S ::= S \cdot S^{\perp}S^{\perp}b$
$S^{\perp}$
$\mathcal{R}1$
$S ::= SS^{\perp} \cdot S^{\perp}b$
$\mathcal{R}1$
$S^{\perp}$
$S ::= SS^{\perp}S^{\perp} \cdot b$
b
$S ::= SS^{\perp}S^{\perp}b\cdot$
$\mathcal{R}2$

Figure 7.15: The IRIA(Γ_S) of Grammar 7.10.



0
$S' ::= \cdot S$
$S ::= \cdot SS^{\perp}S^{\perp}b$
$S ::= \cdot$
$\mathcal{R}2$
1
$S' ::= S\cdot$
$S ::= S \cdot S^{\perp}S^{\perp}b$
$S^{\perp}$
$\mathcal{R}1$
2
$S ::= SS^{\perp} \cdot S^{\perp}b$
$S^{\perp}$
4
$S ::= SS^{\perp}S^{\perp}b\cdot$
b
3
$S ::= SS^{\perp}S^{\perp} \cdot b$

Figure 7.16: The RIA(Γ_S) of Grammar 7.10.

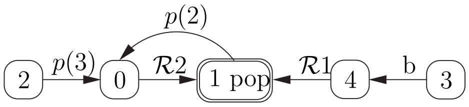
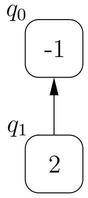


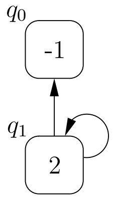
Figure 7.17: The RCA(Γ_S) of Grammar 7.10.

To parse the string b we begin by creating the base node of the RCG, q_0, and add the process $(0, q_0)$ to U_0 and $\mathcal{P}_0$. From state 0 in the RCA we move to state 1 on a transition labelled by $\mathcal{R}2$ and add the process $(1, q_0)$ to U_0. Although state 1 contains a pop action, nothing is done since q_0 does not have any children. We then traverse the edge labelled $p(2)$ back to state 0, create the new RCG node, q_1, with an edge to q_0, and add $(0, q_1)$ to U_0 and $\mathcal{P}_0$.



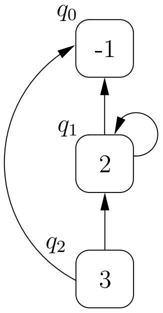
$$\begin{aligned} U_0 &= \{(0, q_0), (1, q_0), (0, q_1)\} \\ \mathcal{P}_0 &= \{(0, q_0), (0, q_1)\} \end{aligned}$$

From state 0 we traverse the $\mathcal{R}2$ edge to state 1 and add the process $(1, q_1)$ to U_0. At this point we can perform the pop action associated with state 1. We find the only child of q_1, q_0, and add $(2, q_0)$ to U_0. Traversing the edge labelled $p(2)$ from state 1 to state 0, we first search $\mathcal{P}_0$ to see if a node labelled 2 has been created during this step of the algorithm. Since q_1 is in $\mathcal{P}_0$ we re-use it and add a new edge from q_1 to itself. However, the new edge on q_1 has created a new path down which the previous pop action could be performed. It is therefore necessary to add $(2, q_1)$ to U_0. The current state of the RCG and the contents of the sets U_0 and $\mathcal{P}_0$ are shown below.



$$\begin{aligned} U_0 &= \{(0, q_0), (1, q_0), (0, q_1), (1, q_1), (2, q_0), \\ &\quad (2, q_1)\} \\ \mathcal{P}_0 &= \{(0, q_0), (0, q_1)\} \end{aligned}$$

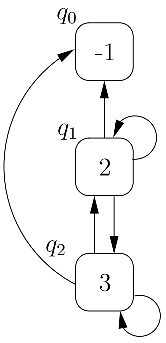
We continue the parse by traversing the edge labelled with the push action, $p(3)$, from state 2 to state 0, create the new RCG node, q_2, labelled 3, with two edges. One edge goes from q_2 to q_0 and the other from q_2 to q_1. We also add $(0, q_2)$ to U_0 and $\mathcal{P}_0$.



$$\begin{aligned} U_0 &= \{(0, q_0), (1, q_0), (0, q_1), (1, q_1), (2, q_0), \\ &\quad (2, q_1), (0, q_2)\} \\ \mathcal{P}_0 &= \{(0, q_0), (0, q_1), (0, q_2)\} \end{aligned}$$

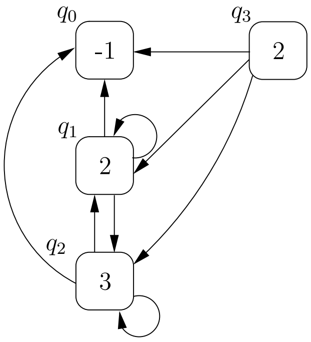
We then traverse the $\mathcal{R}2$ edge to state 1, add $(1, q_2)$ to U_0 and proceed to perform the pop on q_2. We find the children of q_2, q_0 and q_1, and add $(3, q_0)$ and $(3, q_1)$ to U_0. Traversing the edge labelled $p(2)$ from state 1 to state 0, we re-use q_1 and add a new edge from q_1 to q_2. This new edge has created a new path down which the previous pop action could be performed so we add $(2, q_2)$ to U_0. When we perform the push transition, $p3$, from state 2 of the RCA (as a result of $(2, q_2)$ being added to

U_0) we create a new looping edge for q_2. Since a pop we already performed can also be performed down this new edge, we also add $(3, q_2)$ to U_0. That completes the first step of the algorithm. The current state of the RCG and the contents of the sets U_0 and $\mathcal{P}_0$ are shown below.



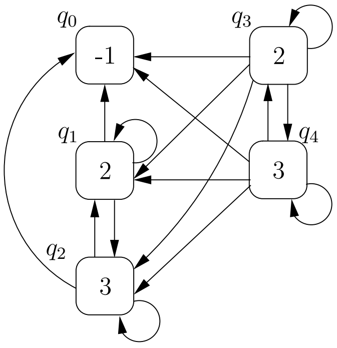
$$\begin{aligned} U_0 &= \{(0, q_0), (1, q_0), (0, q_1), (1, q_1), (2, q_0),\\ &\quad (2, q_1), (0, q_2), (1, q_2), (3, q_0), (3, q_1),\\ &\quad (2, q_2), (3, q_2)\}\\ \mathcal{P}_0 &= \{(0, q_0), (0, q_1), (0, q_2)\} \end{aligned}$$

We then read the next input symbol b, construct the new set $U_1 = \{(4, q_0), (4, q_1), (4, q_2)\}$ and set $\mathcal{P}_1 = \emptyset$. We traverse the $\mathcal{R}1$ edge to state 1, add $(1, q_0), (1, q_1), (1, q_2)$ to U_1 and then perform the associated pop action for q_1 and q_2. We add $(2, q_0), (2, q_1), (2, q_2)$ and $(3, q_0), (3, q_1), (3, q_2)$ to U_1. Traversing the push transition, $p2$, from state 1, we create a new RCG node, q_3, labelled 2 and add edges from it to the nodes q_0, q_1 and q_2.



$$\begin{aligned} U_1 &= \{(4, q_0), (4, q_1), (4, q_2), (1, q_0), (1, q_1),\\ &\quad (1, q_2), (2, q_0), (2, q_1), (2, q_2), (3, q_0),\\ &\quad (3, q_1), (3, q_2)\}\\ \mathcal{P}_1 &= \{(0, q_3)\} \end{aligned}$$

Continuing the parse in this way results in the creation of the final RCG shown below.



$$\begin{aligned} U_1 &= \{(4, q_0), (4, q_1), (4, q_2), (1, q_0), (1, q_1),\\ &\quad (1, q_2), (2, q_0), (2, q_1), (2, q_2), (3, q_0),\\ &\quad (3, q_1), (3, q_2), (0, q_3), (0, q_4), (1, q_3),\\ &\quad (1, q_4)\}\\ \mathcal{P}_1 &= \{(0, q_3), (0, q_4)\} \end{aligned}$$

Since all the input has been consumed and the process $(1, q_0)$ is in U_1, where state 1 is the accepting state of the RCA and q_0 is the RCG's base node, the input string b is accepted.

RIGLR recogniser

```
input data an RCA written as a table $\mathcal{T}$, input string $a_1...a_n$

function PARSER
    $a_{n+1} = \$$, $U_0 = P_0 = \emptyset, \cdots, U_n = P_n = \emptyset$
    create a base node, $q_0$, in the call graph
    create a process node, $u_0$, in $U_0$ labeled $(0, q_0)$ and add $(0, q_0)$ to $P_0$
    for $i = 0$ to $n$ do
        add all the elements of $U_i$ to $A$
        while $A \neq \emptyset$ do
            remove $u = (h, q)$ from $A$
            if $sk \in \mathcal{T}(h, a_{i+1})$ then
                if there is no node labeled $(k, q) \in U_{i+1}$ then
                    create a process node $v$ labeled $(k, q)$
                    add $v$ to $U_{i+1}$
            for each $\mathcal{R}(j, k) \in \mathcal{T}(h, a_{i+1})$ do
                if there is no node labelled $(k, q) \in U_i$ then
                    create a process node $v$ labelled $(k, q)$
                    add $v$ to $A$ and to $U_i$
            if $pop \in \mathcal{T}(h, a_{i+1})$ then
                let $k$ be the label of $q$ and $Z$ be the successor of $q$
                for each $p \in Z$ do
                    if there is no node labelled $(k, q) \in U_{i+1}$ then
                        create a process node $v$ labelled $(k, q)$
                        add $v$ to $A$ and to $U_i$
            for each $p(l, k) \in \mathcal{T}(h, a_{i+1})$ do
                if there is $(k, t) \in P_i$ such that $t$ has label $l$ then
                    add an edge from $t$ to $q$
                    if there is no node labelled $(l, q) \in U_i$ then
                        if there is a node labelled $(f, t)$ in $U_i \setminus A$ and $pop \in \mathcal{T}(f, a_{i+1})$
                            then
                            create a process node $v$ labelled $(l, q)$
                            add $v$ to $A$ and to $U_i$
                else
                    create a node $t$ with label $l$ in the call graph
                    make $q$ a successor of $t$
                    create a process node $v$ labelled $(k, t)$
                    add $v$ to $A$, to $U_i$ and to $P_i$
    if $U_n$ contains a node whose label is $(h_\infty, q_0)$ where $h_\infty$ is an accept state of
        the RCA and $q_0$ is the base node of the call graph **then**
        set *result* = *success*
    **else**
        set *result* = *failure*
    **return** *result*
```

7.6 Reducing the non-determinism in the RCA

We can reduce the amount of non-determinism in the RCA by adding lookahead sets to the reduce, push and pop actions. The lookahead sets are calculated for the reduce and push actions by traversing the $\mathcal{R}$-edges and push edges until a state is reached that has edges leaving it that are labelled by terminal symbols. These terminals are added to the lookahead set of all the edges on the path to the state. If the target state is an accept state then the lookahead set also includes the \$ symbol.

Since the pop action is part of a state, we label the state with a lookahead set which is calculated by finding the lookahead sets of the states that can be reached when the pop action is performed. These states are the targets of the edges labelled by the terminalised symbols in the RIA.

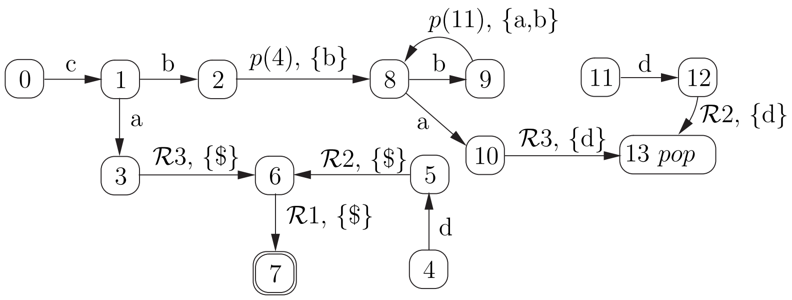
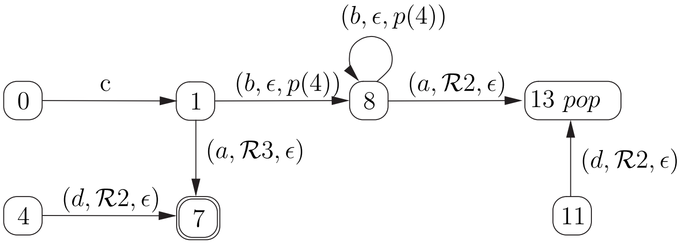


Figure 7.18: The RCA of Grammar 7.6 with lookahead.

The implementation used in PAT does not incorporate lookahead because the RCA are produced by GTB and GTB does not, at present, construct RCA with lookahead.

7.7 Reducing the number of processes in each U_i

The number of processes added to U_i at each step of the RIGLR algorithm can be very large. An approach to reduce both the size of the RCA and the number of processes added at each step of the algorithm is presented in [AHJM01]. It involves 'pre-compiling' and then combining sequences of actions in the RCA. The basic principle is to combine a preceding terminal that can be shifted, with all $\mathcal{R}$-edges, and/or a following push edge. For example, consider the pre-compiled RCA shown in Figure 7.19 that has been constructed from the RCA in Figure 7.18.



$(b, \epsilon, p(4))$
0
c
1
$(b, \epsilon, p(4))$
8
$(a, \mathcal{R}2, \epsilon)$
13 *pop*
$(a, \mathcal{R}3, \epsilon)$
$(d, \mathcal{R}2, \epsilon)$
4
$(d, \mathcal{R}2, \epsilon)$
7
11

Figure 7.19: The reduced RCA of Grammar 7.6 with lookahead.

Although this approach can reduce the size of the RCA, if we want to incorporate lookahead into the RCA, we have to use two symbols of lookahead. This will increase the size of the parse table and hence also the size of the parser. More seriously, if all possible sequences of reductions are composed then the number of new edges in the RCA can be increased from $O(k)$ to $O(2^{k+1})$. An example of this is given in [SJ03b].

However, limited composition has been proposed in [SJ] that guarantees not to increase the size of the RCA. The implementation used in PAT does not incorporate this technique to reduce the size of the sets U_i.

7.8 Generalised regular parsing

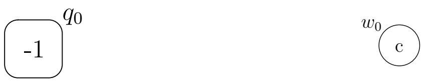
This section introduces the RIGLR parsing algorithm which attempts to find a traversal of an RCA for a string $a_1 \dots a_n$ and constructs a syntactic representation of the string. We begin by providing an informal description of how to build a derivation tree using an example for which the RCA is deterministic. Then we present an efficient way of representing multiple derivations. There is a formal definition of the algorithm at the end of the section.

7.8.1 Constructing derivation trees

We can build a derivation tree for an input string $a_1 \dots a_n$ during a parse by maintaining a sequence of tree nodes, $u_1, \dots, u_p$, constructed at each step of the algorithm. When an edge labelled with a terminal symbol, a, is traversed, we create a parse tree node and append it to the sequence of nodes created thus far. When a $\mathcal{R}_i$ edge is traversed, where rule i is $A ::= x_1 \dots x_k$, we remove nodes, $u_{p-k+1}, \dots, u_p$, from the sequence, create a new node labelled A, with children $u_{p-k+1}, \cdots, u_p$, and append the new node to the end of the sequence. No tree nodes need to be created for the push and pop transitions of the RCA.

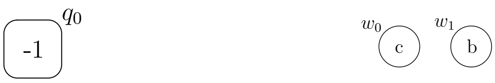
For example, consider the parse of the string *cbad* for Grammar 7.6 and its associated RCA shown in Figure 7.10 (see page 154). We shall maintain the parse tree root nodes in the sequence $\mathcal{W}$.

We begin the parse, as usual, by creating the base node of the RCG and adding the element $(0, q_0)$ to the set U_0 and $\mathcal{P}_0$. We read the first input symbol c, traverse the edge from state 0 to state 1, create the first parse tree node, w_0, labelled c and construct the set $U_1 = \{(1, q_0)\}$.



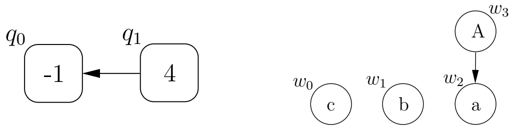
$$\begin{aligned} U_1 &= \{(1, q_0)\} \\ \mathcal{W} &= [w_0] \end{aligned}$$

We continue the parse by reading the next input symbol, b, traversing the edge from state 1, labelled b, to state 2, create the new parse tree node, w_1, and construct the set $U_2 = \{(2, q_0)\}$.



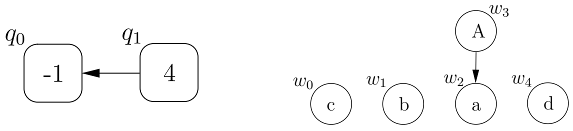
$$\begin{aligned} U_2 &= \{(0, q_0)\} \\ \mathcal{W} &= [w_0, w_1] \end{aligned}$$

The next transition from state 2 is labelled by the push action $p(4)$, so we create the new RCG node, q_1, labelled 4 and add $(8, q_1)$ to U_2. (Recall that no nodes are created in the parse tree as a result of the traversal of a push edge in the RCA.) We then read the third input symbol, a, traverse the edge to state 10, create the parse tree node w_2, labelled a, and construct the set $U_3 = \{(10, q_1)\}$. From state 10, there is a reduction transition labelled $\mathcal{R}_3$ for rule $A ::= a$. We traverse the edge to state 13 and then create the new parse tree node, w_3, labelled A. Since rule 3 has a right hand side of length 1, we remove the last element from the sequence $\mathcal{W}$ and use it to label the child node of w_3. We then add the new root node to the set $\mathcal{W}$, and add $(13, q_1)$ to U_3.



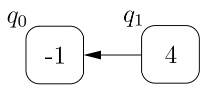
$$\begin{aligned} U_3 &= \{(10, q_1), (13, q_1)\} \\ \mathcal{W} &= [w_0, w_1, w_3] \end{aligned}$$

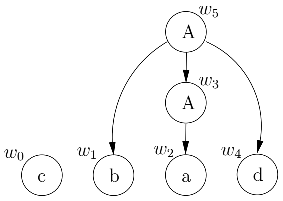
Since state 13 contains a pop action, we find the child, q_0, of the RCG node q_1, and add $(4, q_0)$ to U_3. We read the final input symbol d, create the new parse tree node, w_4, and construct U_4 as shown below.



$$\begin{aligned} U_4 &= \{(5, q_0)\} \\ \mathcal{W} &= [w_0, w_1, w_3, w_4] \end{aligned}$$

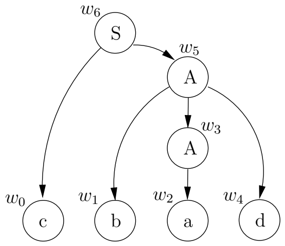
We then traverse the reduction transition, $\mathcal{R}_2$ from state 5. Rule 2 is $A ::= bAd$, so we create the new node, w_5, labelled A, remove the last three parse tree nodes from the sequence $\mathcal{W}$ and add them to w_5 as its children. We also add w_5 to $\mathcal{W}$ and $(6, q_0)$ to U_4.





$$
\begin{aligned}
U_4 &= \{(5, q_0), (6, q_0)\} \\
\mathcal{W} &= [w_0, w_5]
\end{aligned}
$$

We then traverse the reduction transition, $\mathcal{R}1$ from state 6 to state 7. Rule 1 is $S ::= cA$, so we create the new node w_6, labelled S, remove the final two parse tree nodes from the sequence $\mathcal{W}$, and add them to w_6 as its children. Since we have consumed all the input and the process $(7, q_0)$ is in U_4, where 7 is the RCA's accepting state and q_0 is the base node of the RCG, the input string *cbad* is accepted by the parser. The final parse tree is shown below.



In the above example, the RCA is deterministic and hence there is only one derivation tree for the input string. Recall that some parses can have an exponential, or even infinite, number of derivations for a given string. We use Tomita's SPPF representation of multiple parse trees to reduce the amount of space necessary to represent all parse trees. The next section discusses three different approaches that can be used to construct an SPPF of a parse using an RCA.

7.8.2 Constructing an SPPF

There are several different approaches that can be taken to construct an SPPF during a parse of the RIGLR algorithm. A straightforward technique is to store a sequence of SPPF nodes that correspond to the roots of the sub-trees that have been constructed so far with each process (k, q). However, since it is possible to reach a state in the RCA by traversing more than one path, this approach can significantly increase the number of processes constructed at each step of the algorithm [SJ03b].

An alternative approach is to merge the processes that share the same call graph nodes, so as to limit the number of processes that need to be created. Instead of storing the sequence of SPPF nodes directly in a process, we can represent the sequences of SPPF nodes in an *SPPF node graph* – a type of graph structured stack – and replace the sequence of nodes in a process by a single SPPF node graph node.

Unfortunately, as is shown in [SJ03b], this approach leads to spurious derivations being created in the SPPF for certain parses. Another disadvantage of using the SPPF node graph is that the structure needs to be traversed in order to construct the final SPPF, which is likely to affect the order of the parsing algorithm.

A solution proposed in [SJ03b] is to use special SPPF pointer nodes that point to sequences of SPPF nodes within the SPPF. To prevent the spurious derivations from being created, the edges in the call graph are labelled with these special SPPF pointer nodes. This provides a way of only associating the correct derivation sequences with their associated processes.

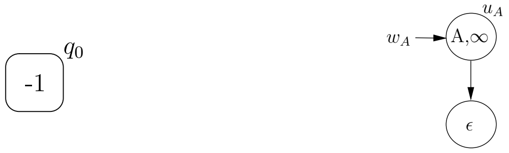
Another problem with the construction of the SPPF is caused if an existing node in the RCG, that has already had a pop applied, has a new edge added to it as a result of a push action. In Example 7.5.1 we show how the recognition algorithm deals with such cases – when a new edge is added to an existing node, all processes in U_i are checked to see if a pop was applied. Unfortunately, this approach does not simply work for the parser because we need to ensure that the correct tree is constructed for any pops that are re-applied. Thus we use the set $\mathcal{P}_i$ to store the SPPF nodes that are associated to a pop's reduction path. When a pop is performed, we add the sequence of SPPF nodes to a set F and store it in the pair (q, F) in $\mathcal{P}_i$. When a push action for a process (h, q, w) is performed, we check to see if $\mathcal{P}_i$ contains an element of the form (q, F). If it does, then a pop has already been performed from state q. We use the sequences of SPPF nodes in F to add the required new processes to U_i that will cause the pops to be re-applied for the correct reduction paths.

Before presenting the formal description of the RIGLR parser we shall work through an example of the construction of an SPPF during a parse using the final approach discussed above. For example, consider the parse of the string *abcc*, with Grammar 7.8 and RCA shown in Figure 7.14 on page 159.

In addition to the sets U_i and $\mathcal{P}_i$ we maintain two additional sets during parsing. The sets $\mathcal{N}$ and $\mathcal{W}$ are used to store the SPPF nodes and special pointer nodes, respectively, that are created at the current step of the algorithm. Recall from Chapter 4 that nodes in an SPPF that are labelled by the same non-terminal and derive the same portion of the input string can be merged. To achieve this efficiently, we label each SPPF node with a pair (x, j), where j is an integer representing the start position within the input string that the yield of the sub-graph (x, j) derives. (The set $\mathcal{N}$ is used to reduce the number of SPPF nodes that need to searched to find the ones that can be packed. It also allows SPPF nodes to be labelled by the pair (x, j) since all nodes in the set will have been constructed at the current step of the algorithm.)

Furthermore, the SPPF of a specific ϵ-rule can be shared throughout a parse whenever a reduction is done for that rule. To avoid creating redundant instances of an ϵ-SPPF we shall create all ϵ-SPPF's at the start of the algorithm and use them whenever necessary.

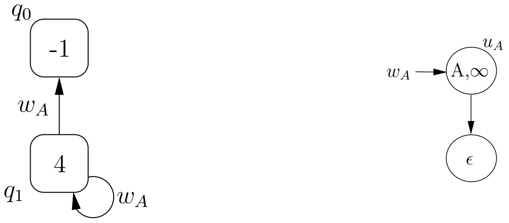
Since Grammar 7.8 contains the ϵ-rule $A ::= \epsilon$, we build the tree, u_A, with pointer w_A. We begin the parse by creating the base node of the RCG and adding the process $(0, q_0, \epsilon)$ to U_i.



$$
\begin{aligned}
U_0 &= \{(0, q_0, \epsilon)\} \\
\mathcal{P}_0 &= \{(q_0, \emptyset)\} \\
\mathcal{N} &= \{\} \\
\mathcal{W} &= \{\}
\end{aligned}
$$

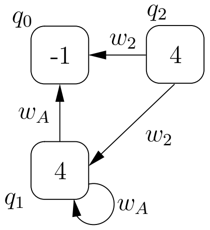
We then traverse the edge labelled $\mathcal{R}4$ from state 0 to state 3, add the process $(3, q_0, w_A)$ to U_0 and then traverse the push edge labelled $p(4)$ back to state 0. The push action results in the new RCG node, q_1, with an edge labelled w_A to q_0, being created and $(0, q_1, \epsilon)$ being added to U_0 and $(q_1, \emptyset)$ to $\mathcal{P}_0$. The SPPF is not modified.

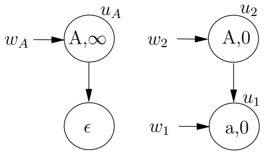
We traverse the $\mathcal{R}4$ edge again for process $(0, q_1, \epsilon)$, add $(3, q_1, w_A)$ to U_0 and then traverse the push transition $p(4)$. This results in a new edge, labelled w_A, being added to the RCG as a loop on q_1.



$$
\begin{aligned}
U_0 &= \{(0, q_0, \epsilon), (3, q_0, w_A), (0, q_1, \epsilon), (3, q_1, w_A)\} \\
\mathcal{P}_0 &= \{(q_0, \emptyset), (q_1, \emptyset)\} \\
\mathcal{N} &= \{\} \\
\mathcal{W} &= \{\}
\end{aligned}
$$

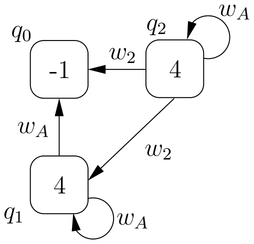
We continue the parse by reading the next input symbol, a, and then traverse the edge from state 0 to state 2. We create the new SPPF node, u_1, labelled $(a, 0)$, add a pointer node w_1 to u_1 and construct $U_1 = \{(2, q_0, w_1), (2, q_1, w_1)\}$. For each of these processes we traverse the edge labelled $\mathcal{R}3$ to state 3, create the SPPF node, u_2, labelled $(A, 0)$ with pointer node w_2, and add the processes $(3, q_0, w_2)$ and $(3, q_1, w_2)$ to U_1. We add u_1 as the child of u_2 since we created the new SPPF node from the processes $(2, q_0, w_1), (2, q_1, w_1)$. From state 3 we traverse the edge $p(4)$ to state 0, create the new RCG node, q_2, and two edges labelled w_2 from q_2 to q_0 and q_1. The process $(0, q_2, \epsilon)$ is also added to U_1 and $(q_2, \emptyset)$ to $\mathcal{P}$.

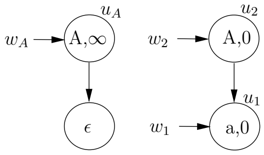




$$\begin{aligned}
U_1 &= \{(2, q_0, w_1), (2, q_1, w_1), (3, q_0, w_2), (3, q_1, w_2), (0, q_2, \epsilon)\} \\
\mathcal{P}_1 &= \{(q_2, \emptyset)\} \\
\mathcal{N} &= \{u_1, u_2\} \\
\mathcal{W} &= \{w_1, w_2\}
\end{aligned}$$

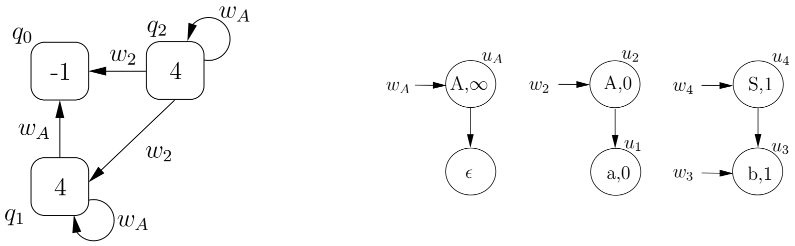
We then traverse the edge $\mathcal{R}4$ from state 0 to state 3 for process $(0, q_2, \epsilon)$, followed by the push transition back to state 0. This results in the process $(3, q_2, w_A)$ being added to U_1, and the edge labelled w_A being added to the RCG from q_2 to itself.





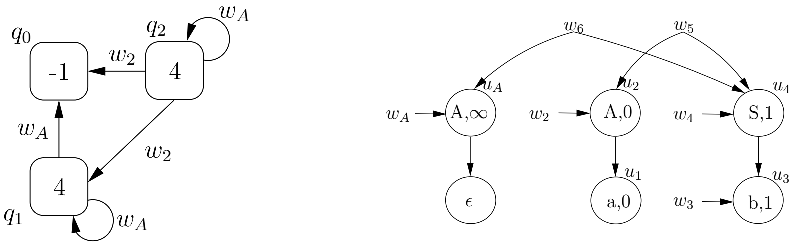
$$\begin{aligned}
U_1 &= \{(2, q_0, w_1), (2, q_1, w_1), (3, q_0, w_2), (3, q_1, w_2), (0, q_2, \epsilon), \\
&\quad (3, q_2, w_A)\} \\
\mathcal{P}_1 &= \{(q_2, \emptyset)\} \\
\mathcal{N} &= \{u_1, u_2\} \\
\mathcal{W} &= \{w_1, w_2\}
\end{aligned}$$

Traversing the edge labelled b from state 0, we create the new SPPF node, u_3, labelled $(b, 1)$, with pointer node w_3, and construct $U_2 = \{(1, q_2, w_3)\}$. We then traverse the $\mathcal{R}2$ edge to state 5, create the SPPF node u_4, labelled $(S, 1)$, with pointer node w_4 and update the relevant sets as shown below.



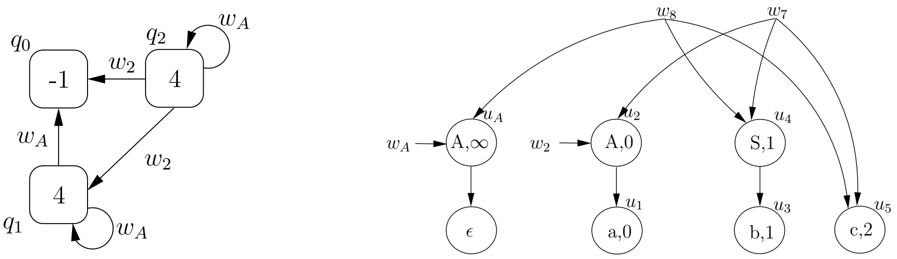
$$
\begin{aligned}
U_2 &= \{(1, q_2, w_3), (5, q_2, w_4)\} \\
\mathcal{P}_2 &= \{\} \\
\mathcal{N} &= \{u_3, u_4\} \\
\mathcal{W} &= \{w_3, w_4\}
\end{aligned}
$$

Performing the pop action associated with state 5, for process $(5, q_2, w_4)$, we first find the children of q_2, q_0, q_1, q_2. Then we create two new pointer nodes w_5 and w_6, with edges to the children of the pointers of the popped edges in the RCG and w_4. For each of the new pointers we also add the processes $(4, q_0, w_5), (4, q_1, w_5)$ and $(4, q_2, w_6)$ to U_2.



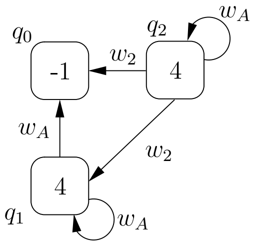
$$
\begin{aligned}
U_2 &= \{(1, q_2, w_3), (5, q_2, w_4), (4, q_0, w_5), (4, q_1, w_5), (4, q_2, w_6)\} \\
\mathcal{P}_2 &= \{\} \\
\mathcal{N} &= \{u_3, u_4\} \\
\mathcal{W} &= \{w_3, w_4, w_5, w_6\}
\end{aligned}
$$

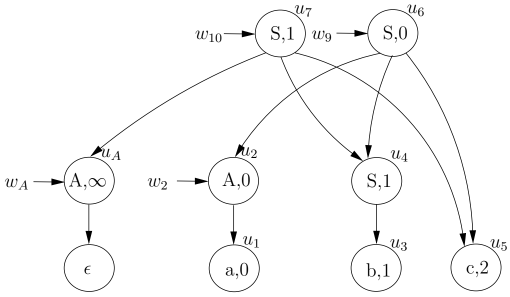
We continue the parse by reading the next input symbol c and then create the new SPPF node, u_5, labelled $(c, 3)$, with two pointer nodes w_7 and w_8. We construct $U_3 = \{(6, q_0, w_7), (6, q_1, w_7), (6, q_2, w_8)\}$, $\mathcal{N} = \{u_5\}$ and $\mathcal{W} = \{w_7, w_8\}$.



$$\begin{aligned} U_3 &= \{(6, q_0, w_7), (6, q_1, w_7), (6, q_2, w_8)\} \\ \mathcal{P}_3 &= \{\} \\ \mathcal{N} &= \{u_5\} \\ \mathcal{W} &= \{w_7, w_8\} \end{aligned}$$

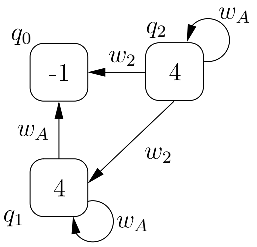
From state 6 we the traverse the $\mathcal{R}1$ edge to state 5 and create two new SPPF nodes u_9 and u_{10}, labelled $(S, 0)$ and $(S, 1)$, respectively. The state of the parser is shown below.

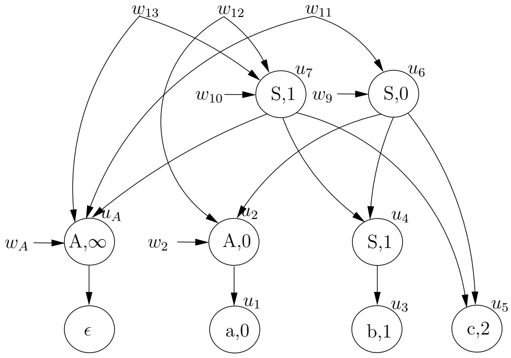




$$\begin{aligned} U_3 &= \{(6, q_0, w_7), (6, q_1, w_7), (6, q_2, w_8), (5, q_0, w_9), (5, q_1, w_9), (5, q_2, w_{10})\} \\ \mathcal{P}_3 &= \{\} \\ \mathcal{N} &= \{u_5, u_6, u_7\} \\ \mathcal{W} &= \{w_7, w_8, w_9, w_{10}\} \end{aligned}$$

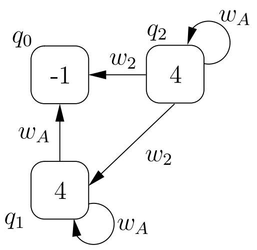
Performing the pop action associated with state 5, for processes $(5, q_1, w_9)$ and $(5, q_2, w_{10})$, we first find the children of q_1 and q_2, q_0, q_1 and q_0, q_1, q_2. Then we create three new pointer nodes w_{11}, w_{12} and w_{13}, with edges to the children of the pointers of the popped edges in the RCG and w_9 and w_{10} respectively. The state of the parser after the pop is complete is shown below.

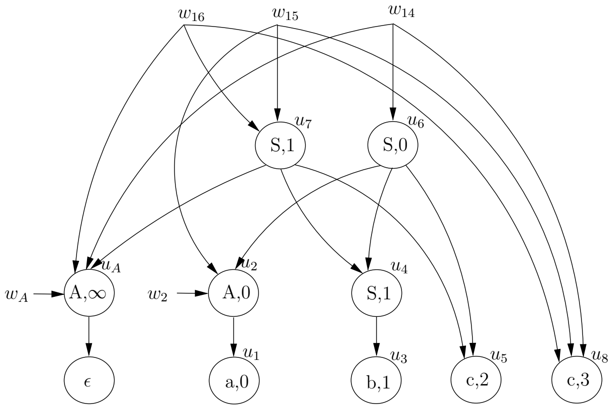




$$
\begin{aligned}
U_3 &= \{(6, q_0, w_7), (6, q_1, w_7), (6, q_2, w_8), (5, q_0, w_9), (5, q_1, w_9), (5, q_2, w_{10}) \\
&\quad (4, q_0, \{w_{11}, w_{12}\}), (4, q_1, \{w_{11}, w_{12}\}), (4, q_2, w_{13}), \} \\
\mathcal{P}_3 &= \{\} \\
\mathcal{N} &= \{u_5, u_6, u_7\} \\
\mathcal{W} &= \{w_7, w_8, w_9, w_{10}, w_{11}, w_{12}, w_{13}\}
\end{aligned}
$$

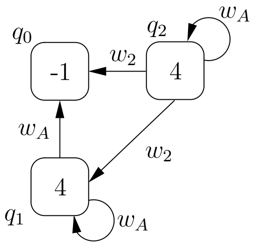
Next we read the final input symbol c, create u_8 labelled $(c, 3)$ and three new pointer nodes w_{14}, w_{15} and w_{16} in the SPPF. We traverse the edge from state 4 to state 6 and construct U_4.

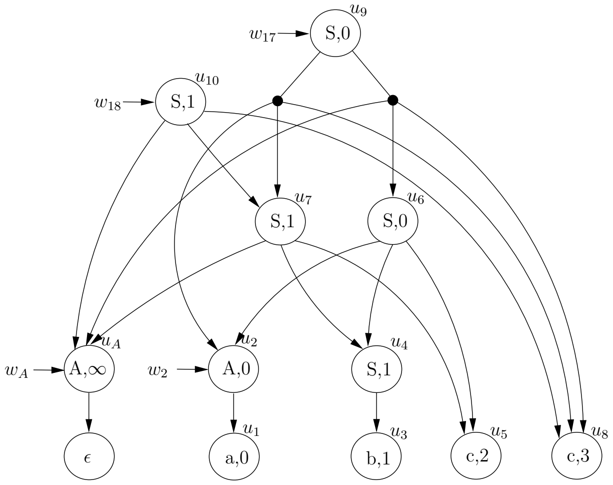




$$
\begin{aligned}
U_4 &= \{(6, q_0, \{w_{14}, w_{15}\}), (6, q_1, \{w_{14}, w_{15}\}), (6, q_2, w_{16})\} \\
\mathcal{P}_4 &= \{\} \\
\mathcal{N} &= \{u_8\} \\
\mathcal{W} &= \{w_{14}, w_{15}, w_{16}\}
\end{aligned}
$$

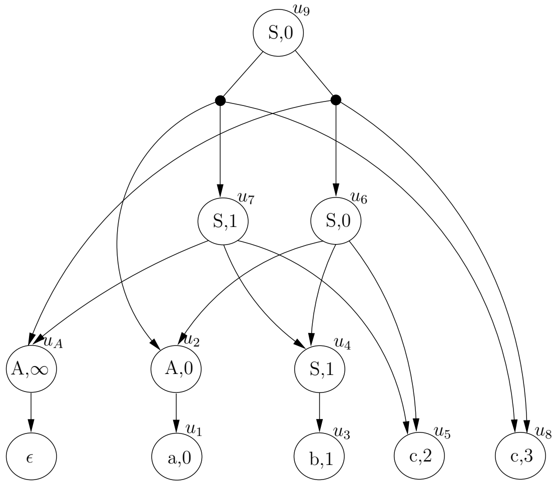
For each of the processes in U_4 we traverse the $\mathcal{R}1$ edge from state 6 in the RCA. For processes $(6, q_0, \{w_{14}, w_{15}\})$ and $(6, q_1, \{w_{14}, w_{15}\})$ we create a new SPPF node, u_9, labelled $(S, 0)$, with a pointer node w_{17}. Since there are two pointer nodes w_{14} and w_{15} associated with both processes, we create two packing nodes below u_9, one with edges to the children of w_{14} and the other with edges to the children of w_{15}. For $(6, q_2, w_{16})$ we create the new SPPF node, u_{10}, labelled $(S, 1)$ with pointer node w_{18} and edges to the children of w_{16}.





$$
\begin{aligned}
U_4 &= \{(6, q_0, \{w_{14}, w_{15}\}), (6, q_1, \{w_{14}, w_{15}\}), (6, q_2, w_{16}), (5, q_0, w_{17}), \\
&\quad (5, q_1, w_{17}), (5, q_2, w_{18})\} \\
\mathcal{P}_4 &= \{\} \\
\mathcal{N} &= \{u_9, u_{10}\} \\
\mathcal{W} &= \{w_{17}, w_{18}\}
\end{aligned}
$$

Since the process $(5, q_0, w_{17})$ is in U_4, and state 5 is the accept state of the RCA, and q_0 is the base node of the RCG, the string *abcc* is accepted. We make the root node of the SPPF the node pointed to by w_{17}. The final SPPF, without any pointer nodes is shown below.



RIGLR parser

input data an RCA written as a table $\mathcal{T}$, input string $a_1 ... a_n$

```
function PARSER
    $a_{n+1} = \$$, $U_0 = P_0 = \emptyset, \ldots, U_n = P_n = \emptyset, \mathcal{W} = \mathcal{W}' = \emptyset$
    create a base node, $q_0$, in the RCG
    create a process $(0, q_0, \epsilon)$ and add to $U_0$ and $(q_0, \emptyset)$ to $P_0$
    for each rule $A ::= \epsilon$ do
        create an SPPF node $v_A$ labeled $(A, \infty)$ with child node labeled $\epsilon$ and a
            pointer $w_A$ to $v_A$
    for $i = 0$ to $n$ do
        add all the elements of $U_i$ to $\mathcal{A}$
        set $\mathcal{W}$ to be $\mathcal{W}''$
        set $\mathcal{N} = \emptyset$ and $\mathcal{W}'' = \emptyset$
        /* $\mathcal{W}''$ holds the pointer nodes needed for the next step */
        while $\mathcal{A} \neq \emptyset$ do
            remove $u = (h, q, w)$ from $\mathcal{A}$
            let $v_d, \ldots, v_1$ be the sequence of nodes pointed to by $w$ in left to right
                order
            if $sk \in \mathcal{T}(h, a_{i+1})$ then
                if there is no SPPF node $u$ labelled $(a_{i+1}, i)$ in $\mathcal{N}$ then
                    create one
                if there is no pointer node $w'$ in $\mathcal{W}''$ with children $v_d, \ldots, v_1, u$ then
                    create one
                if there is no process $(k, q, w') \in U_{i+1}$ then
                    add $(k, q, w')$ to $U_{i+1}$
            for each $\mathcal{R}(j, k) \in \mathcal{T}(h, a_{i+1})$ such that (length of $j$) $\leq d$
                let rule $j$ be $A ::= x_l \cdots x_1$ do
                if $l = 0$ then let $u = v_A$
                else
                    let $(x_i, f_i)$ be the label of $v_i$ and let $f = min\{f_i \mid l \leq i \leq d\}$
                    if there is no SPPF node $u$ in $\mathcal{N}$ labelled $(A, f)$ then
                        create one
                    if $u$ does not have $v_l, \ldots, v_1$ as a sequence of children then
                        add $v_l, \ldots, v_1$ as a sequence of children of $u$.
                if there is no node $w'$ in $\mathcal{W}$ whose children are $v_d, \ldots, v_{l+1}, u$ then
                    create one
                if there is no process $(k, q, w') \in U_i$ then
                    add $(k, q)$ to $\mathcal{A}$ and to $U_i$
            if $pop \in \mathcal{T}(h, a_{i+1})$ then
                let $k$ be the label of $q$
                add $w$ to $F$ where $(q, F_q) \in P_i$
                let $\mathcal{Z}$ contain the pairs $(p, w'')$ where $(q, p)$ is an edge with label $w''$
```

```
      **for each** $(p, w'') \in \mathcal{Z}$ **do**
        let $v_b, v_{b-1}, \ldots, v_{d+1}$ be the children of $w''$ in left to right order
        **if** there is no node $w'$ in $\mathcal{W}$ with children $v_b, \ldots, v_{d+1}, v_d, \ldots, v_1$
          **then**
          create one
        **if** there is no process $(k, p, w') \in U_{i+1}$ **then**
          add $(k, p, w')$ to $\mathcal{A}$ and to $U_i$
    **for each** $p(l, k) \in \mathcal{T}(h, a_{i+1})$ **do**
      add $w$ to $\mathcal{W}''$
      **if** there is $(t, F_t) \in P_i$ such that $t$ has label $l$ **then**
        **if** there is no edge from $t$ to $q$ with label $w$ **then**
          add an edge from $t$ to $q$ labelled $w$
          **for each** $y \in F_t$ **do**
            let $v'_b, \ldots, v'_1$ be the children of $y$ in left to right order
            **if** there is no node $w'$ in $\mathcal{W}$ with children $v_d, \ldots, v_1, v'_b, \ldots, v'_1$
              **then**
              create one
            **if** there is no process $(k, t, w') \in U_i$ **then**
              add $(k, t, w')$ to $\mathcal{A}$ and to $U_i$
      **else**
        create a node $t$ with label $l$ in the RCG
        create an edge $(q, t)$ labelled $w$
        create a process node $v$ labelled $(k, t, \epsilon)$
        add $v$ to $\mathcal{A}$ and $U_i$
        add $(t, \emptyset)$ to $P_i$
**if** $U_n$ contains a node whose label is $(h_\infty, q_0, w_\infty)$ where $h_\infty$ is an accept state of
  the RCA, $q_0$ is the base node of the RCG and $u_\infty$ is the RCG node labelled
  $(S, 0)$ **then**
  set *result* = *success*
**else**
  set *result* = *failure*
**return** *result*
```

7.9 Summary

In this chapter we examined the role of the stack in bottom-up parsers and described how to construct the automata that can be used by the RIGLR recognition and parsing algorithms to parse with less stack activity than other GLR parsing techniques.

Chapter 10 contains the results of several experiments that highlight the performance of the RIGLR recognition algorithm compared to the other GLR algorithms.

Chapter 8

Other approaches to generalised parsing

This chapter looks at other approaches to generalised parsing and compares them to the techniques studied in detail in this thesis.

8.1 Mark-Jan Nederhof and Janos J. Sarbo

Tomita's Algorithm 2 can fail to terminate when parsing strings in hidden-left recursive grammars (see Chapter 4). Farshi extends Tomita's Algorithm 1 to parse all grammars by introducing cycles into the GSS and performing extra searching during parsing. In [NS96], Nederhof and Sarbo present a new type of automaton, the ϵ-LR DFA, that allows Tomita's Algorithm 2 to correctly parse strings in all context-free grammars. This section provides an overview of their approach.

It is claimed in [NS96] that the cycles in the GSS, introduced by Farshi, complicate garbage collection and prevent the use of memo-functions presented by Leermakers et. al., [Lee92a]. Since it is hidden-left recursive rules that cause Tomita's Algorithm 2 to fail to terminate, a straightforward approach to deal with such grammars is to remove any non-terminals that derive ϵ by transforming them with the standard ϵ-removal algorithm [AU73]. Obviously this also removes any hidden-left recursion. Of course, if all the ϵ-rules are removed, the grammar will not contain any right nullable rules either, so Tomita's Algorithm 1 can also be used correctly.

The drawback of the standard ϵ-removal process is that it can significantly increase the number of grammar rules and hence the size of the LR DFA. In principle, since there are more DFA states the GSS may be bigger, so the run time could be affected as well. As far as we know there have been no studies on the comparative GSS size for grammars before and after ϵ removal.

In an attempt to reduce the size of the automaton, certain states in the automaton are merged in a similar way to Pager's [Pag70] parse table compression technique.

For example, consider Grammar 8.1 and its associated DFA shown in Figure 8.1.

$$\begin{aligned} S' &::= S \\ S &::= aAd \\ A &::= Bc \\ B &::= b \\ B &::= \epsilon \end{aligned} \tag{8.1}$$

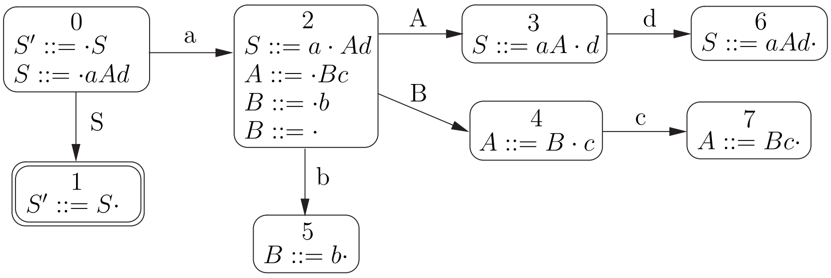


Figure 8.1: The LR(0) DFA for Grammar 8.1.

By transforming Grammar 8.1 with the ϵ-removal algorithm, Grammar 8.2 is constructed. For the purpose of clarity, we have used non-terminals of the form $[A]$ to indicate that the non-terminal A has been removed from the grammar by the ϵ-removal transformation.

$$\begin{aligned} S' &::= S \\ S &::= aAd \\ A &::= Bc \\ A &::= [B]c \\ B &::= b \end{aligned} \tag{8.2}$$

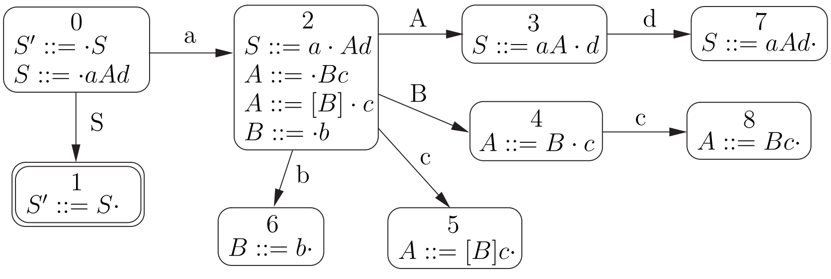


Figure 8.2: The LR(0) DFA for Grammar 8.2.

To reduce the number of states in the DFA of a grammar that has had the ϵ-removal transformation applied, we can incorporate the removal of ϵ-rules into the closure function used during the construction of the DFA. We consider rules that are derived from the same *basic* item to be equivalent. For example, in Figure 8.2 the items $(A ::= Bc\cdot)$ and $(A ::= [B]c\cdot)$ are considered to be the same and hence states 5 and 7 can be merged and a new edge, labelled c, added from state 5.

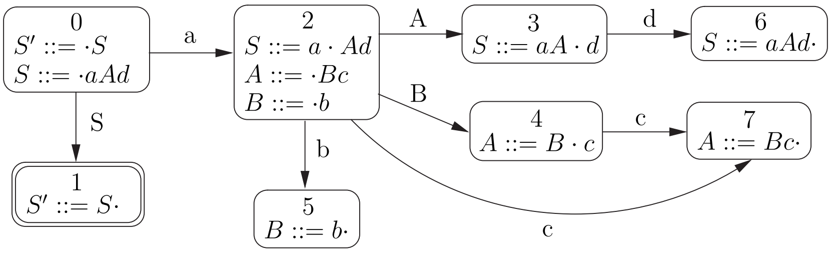
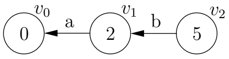


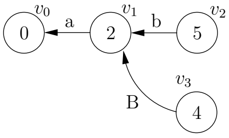
Figure 8.3: The ϵ-LR(0) DFA for Grammar 8.2.

Although the merging of DFA states in this way can reduce the number of states, it breaks one of the fundamental properties of DFA's that are used by parsers – if a state that contains a reduction action is reached, then it is guaranteed that the reduction is applicable. As a result, when a reduction is performed by a parser that uses the ϵ-LR DFA, it is necessary to check each of the symbols that are popped off the stack, to ensure that the reduction is applicable at that point of the parse.

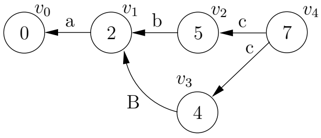
For example, consider the parse of the string $abcd$ using the ϵ-LR(0) DFA in Figure 8.3. We begin by creating the node, v_0, labelled by the start symbol of the DFA and then perform two consecutive shifts for the first two input symbols ab.



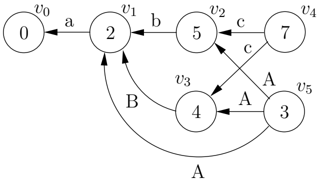
From state 5 we can perform a reduce for rule $B ::= b$, so we trace back a path of length 1 from v_2, checking that the symbol on the edge matches the rule, to state v_1 and create the new node, v_3 labelled 4.



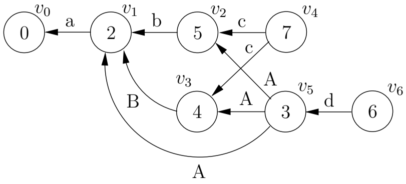
At this point we can perform the shift on c from state 4 in the DFA to state 7. However, because of the previous reduction the shift is now also applicable from v_2. We create the new node v_4 and add edges to both v_3 and v_2.



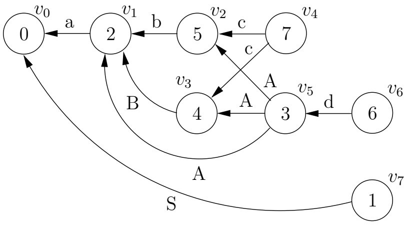
From state 7 there is the reduction, $A ::= Bc\cdot$. However, recall that state 7 was created by merging states 5 and 8 in the DFA in Figure 8.2, so there are two reductions that are valid at this point. One where $B \Rightarrow \epsilon$ and the other where $B \Rightarrow b$. Therefore we trace back two types of path; one of length 1, where we can pop c off the stack and another of length 2, where we pop Bc. Consequently we construct the GSS show below.



Next we can read the final input symbol d and perform the shift from state 3 to state 6. We create the new node v_6 and create the edge from v_6 to v_5.



From state 6 we perform the reduction $S ::= aAd$ by tracing back paths of length 3 from v_6. However, although three different paths can be traversed, ($[v_6, v_5, v_2, v_1]$, $[v_6, v_5, v_3, v_1]$, $[v_6, v_5, v_1, v_0]$) only one is applicable because of the symbols labeling the traversed edges. The final GSS is shown below.



Another solution, which only causes a quadratic increase in the number of rules, is to use the hidden-left recursion removal algorithm. However, a grammar that has been transformed in this way can only be parsed by Tomita's Algorithm 2, since Algorithm 1 has a problem with right nullable rules. As we shall see in Chapter 10 Algorithm 2 is less efficient than Algorithm 1 (or indeed RNGLR).

8.2 James R. Kipps

Tomita's Algorithm 2 can fail to terminate when parsing strings in hidden-left recursive grammars (see Chapter 4). In [Kip91], Kipps shows that the worst case asymptotic complexity of the algorithm, for grammars without hidden-left recursion, is $O(n^{k+1})$. However, his proof does not take this into consideration and hence must be incorrect. None of the grammars used in the experimental section of [Kip91] contain hidden-left recursion and hence successfully terminate.

Kipps makes a modification to Tomita's algorithm which, he claims, makes it achieve $O(n^3)$ worst case time complexity for all context-free grammars. Kipps changes the formal definition used by Tomita in [Tom85]. The REDUCER and ϵ-REDUCER are combined into one function and the new ANCESTORS function is used to abstract the search for target nodes of a reduction. Also, instead of defining the algorithm to create sub-frontiers for nullable reductions the concept of a clone vertex is introduced.

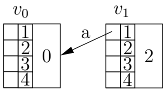
Although the notation and layout of the algorithm presented in [Kip91] differs somewhat from Tomita's Algorithm 2 we believe them to be equivalent. Kipps' algorithm also fails to terminate on grammars with hidden-left recursion and thus cannot be $O(n^3)$. The proof that his algorithm is cubic is flawed in the same way that his proof that Tomita's algorithm is $O(n^{k+1})$.

Although Kipps' proof is flawed the observations on the root of the algorithms complexity are valid – it is the ANCESTORS function that traces back reduction paths during a reduction that contributes to the complexity of the algorithm. Only the ANCESTORS function uses i^k steps. However, ANCESTORS can only ever return at most i nodes and there are at most i nodes between a node in U_i and its ancestors. So, for i^k steps to be performed in ANCESTORS some paths must be traversed more than once. Kipps improves the performance of Tomita's algorithm by constructing an *ancestors table* that allows the fast look-up of nodes at a distance of k.

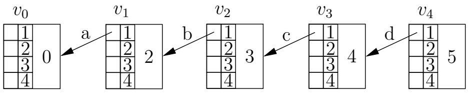
In Kipps' algorithm a state node v is represented as a triple $\langle i, s, l \rangle$ where i is the level the state is in, s is the state number labeling the node and l is the ancestor field that stores the portion of the ancestor table for v. In the algorithm the ancestor field of a node consists of sets of tuples $\langle k, l_k \rangle$ where l_k are the set of ancestor (or target) nodes at a length of k from node v. In our example we draw the GSS nodes with a two dimensional ancestor table, representing the portion of the ancestor table for the node, on the left and the state number on the right. We do not label the level in the node since it is clear by the position of the node. To highlight how the algorithm operates we use dotted arrows for the edges added by the ANCESTORS function and solid arrows to represent the final edge of a reduction.

The algorithm dynamically generates the ancestor table as reductions are performed.

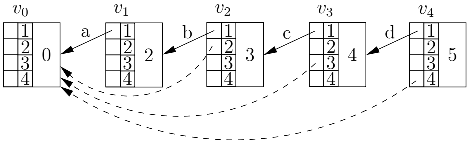
Before we present the formal specification of Kipps' algorithm we demonstrate its operation using an example. Recall Grammar 6.5 and its associated DFA in Figure 6.6, shown on page 129. To parse the string *abcd* we begin by creating the new node v_0 in U_0. Since there is a shift to state 2 from state 0 on the next input symbol a, we create the new node v_1, labelled 2, in U_1. We add an entry in the ancestor field of v_1 to show that v_0 is an ancestor of v_1 at a distance of one. We represent this in the diagram by adding an edge from the ancestor table of v_1 in position 1 to v_0.



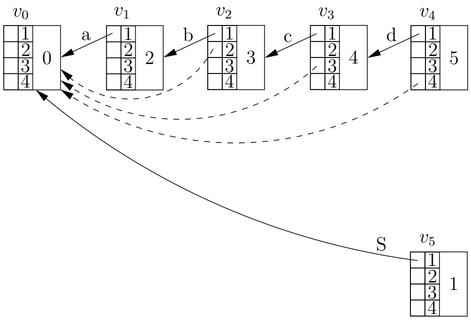
We continue in this way shifting the next three input symbols and creating the nodes v_2, v_3, v_4 labelled 3,4 and 5 respectively.



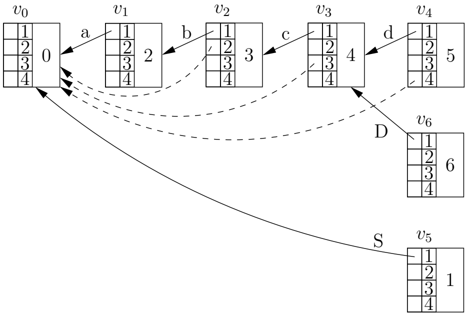
Processing v_4 we find a reduce/reduce conflict in state 5 for the current lookahead symbol \$. First we perform the reduction on rule $S ::= abcd$ by calling the REDUCER function. We find the target node of the reduction by calling the ANCESTORS function. Since there is not a node on a distance of 4 from v_4, we find node v_3 at a distance of 1 and call the ANCESTORS function recursively from v_3 and a length of three. We repeat the recursion until the length reaches 0 at which point we return the node reached. As the recursion returns to each node on the path between v_4 and v_0 we update the ancestor table by adding an edge to the returned target node of the reduction that is getting passed back through the recursion.



The ANCESTORS function returns v_0 as the node at a distance of 4 from v_4. Since no node exists in the frontier of the GSS that is labelled by the goto state of the reduction we create the new node v_5 with an edge from position one in the ancestor table to v_0.

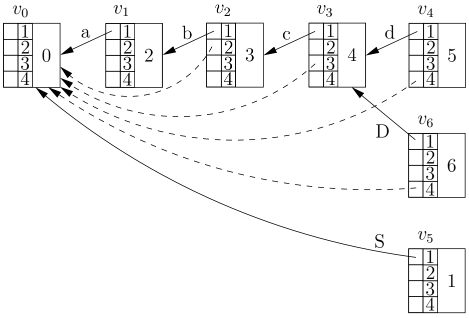


We continue by processing the second reduction, $D ::= d$, of the reduce/reduce conflict encountered in v_4. The reduction is of length one and since there is already an edge in the ancestor table of v_4 the ANCESTORS function returns v_3 without performing any recursion. We create the new node v_6, labelled 6, in U_4 and add an edge in its ancestor table to v_3 in position one.



When we process v_5 we find the accept action in its associated parse table entry. However, since v_6 has not yet been processed, the termination of the parse is delayed. Processing v_6 we find the reduction on rule $S ::= abcD$. We use the ANCESTORS function to find the target node of the reduction on a path of length four from v_6. Since we have already performed a reduction from v_4 that shares some of the path used by this reduction, the ancestor table in v_3 contains an edge in position 3 to v_0. Instead of continuing the recursion, the ANCESTORS function returns v_0 as the node at the end of the path of length four from v_6 without tracing the entire reduction path.

Since v_5 is labelled by the goto state of the reduction and has an edge to the target node v_0, the GSS remains unchanged and the parse terminates in success.



Kipps' $O(n^3)$ recognition algorithm

P is the set that maintains the nodes that have already been processed.

```
function REC($x_1 \cdots x_n$)
    let $x_{n+1} := \$$
    let $U_i := [\,]$     $(0 \le i \le n)$
    let $U_0 := [\langle 0, S_0, \emptyset \rangle]$
    **for** $i$ from 1 to $n+1$ **do**
        let $P := [\ ]$
        **for** $\forall v = \langle i-1, s, l \rangle$ s.t. $v \in U_{i-1}$ **do**
            let $P := P \circ [v]$
            **if** $\exists$ ‘sh $s'$’ $\in \textsc{Actions}(s, x_i)$ **then** $\textsc{Shift}(v, s')$
            **for** $\forall$ ‘re $p$’ $\in \textsc{Actions}(s, x_i)$ **do** $\textsc{Reduce}(v, p)$
            **if** ‘acc’ $\in \textsc{Actions}(s, x_i)$ **then** *accept*
        **if** $U_i$ is empty **then** *reject*

**function** $\textsc{Shift}(v, s)$
    **if** $\exists v' = \langle i, s, a \rangle$ s.t. $v' \in U_i \land \langle 1, l \rangle \in a$ **then**
        let $l := l \cup \{v\}$
    **else**
        let $U_i := U_i \circ [\langle i, s, [\langle 1, \{v\} \rangle] \rangle]$

**function** $\textsc{Reduce}(v, p)$
    **for** $\forall v'_1 = \langle j', s', a'_1 \rangle$ s.t. $v'_1 \in \textsc{Ancestors}(v, \bar{p})$ **do**
        let ‘go $s''$’ $:= \textsc{Goto}(s', D_p)$
        **if** $\exists v'' = \langle i-1, s'', a'' \rangle$ s.t. $v'' \in U_{i-1} \land \langle 1, l'' \rangle \in a''$ **then**
            **if** $v'_1 \in l''$ **then**
                *do nothing (ambiguous)*
            **else**
                **if** $\exists v'_2 = \langle j', s', a'_2 \rangle$ s.t. $v'_2 \in l''$ **then**
                    let $v''_c := \langle i-1, s'', a''_c \rangle$ s.t. $a''_c = [\langle 1, \{v'_1\} \rangle]$
                    **for** $\forall$ ‘re $p$’ $\in \textsc{Actions}(s'', x_i)$ **do** $\textsc{Reduce}(v''_c, p)$
                    let $l_{k_1} := l_{k_1} \cup l_{k_2}$ s.t. $\langle k, l_{k_1} \rangle \in a'' \land \langle k, l_{k_2} \rangle \in a''_c \quad (k \geq 2)$
                **else**
                    let $l'' := l'' \cup \{v'_1\}$
                    **if** $v'' \in P$ **then**
                        let $v''_c := \langle i-1, s'', a''_c \rangle$ s.t. $a''_c = [\langle 1, \{v'_1\} \rangle]$
                        **for** $\forall$ ‘re $p$’ $\in \textsc{Actions}(s'', x_i)$ **do** $\textsc{Reduce}(v''_c, p)$
                        let $l_{k_1} := l_{k_1} \cup l_{k_2}$ s.t. $\langle k, l_{k_1} \rangle \in a'' \land \langle k, l_{k_2} \rangle \in a''_c \quad (k \geq 2)$
        **else**
            let $U_{i-1} = U_{i-1} \circ [\langle i-1, s'', \{v'_1\} \rangle]$

**function** $\textsc{Ancestors}(v = \langle j, s, a \rangle, k)$
    **if** $k = 0$ **then**
        **return** $(\{v\})$
    **else if** $\exists \langle k, l_k \rangle \in a$ **then**
```

```
        return ($l_k$)
    else
        let $l_k := U_{v' \in l_1 | \langle 1, l_1 \rangle \in a}$ Ancestors($v', k-1$)
        $a := a \cup \{\langle k, l_k \rangle\}$
        return ($l_k$)
```

In a similar way to the BRNGLR algorithm Kipps' approach trades space for time, but the two algorithms use different techniques to achieve this. In the conclusion of his paper Kipps admits that although his approach produces an asymptotically more efficient parser, the overheads associated with his method do not justify the improvements. He argues that the ambiguity of grammars in *real* applications is restricted by the fact that humans must be able to understand them.

8.3 Jay Earley

One of the most popular generalised parsing algorithms is Earley's algorithm [Ear67, Ear68]. First described in 1967, this approach has received a lot of attention over the last 40 years. In this section we briefly discuss the operation of the algorithm and highlight some of its associated problems.

Earley's algorithm uses a grammar G to parse an input string $X_1 \dots X_n$ by dynamically constructing sets of items similar to those used by the LR parsing algorithm. The only difference between Earley's items and the standard LR items used in the DFA construction is the addition of a pointer back to the set containing the base item of the rule.

For example consider the parse of the string *aab* with Grammar 6.1 shown on page 118. We begin by initialising the set S_0 with the item ($S ::= \cdot S$ 0). We predict the rule for S and add the items ($S ::= \cdot aSB$ 0) and ($S ::= \cdot b$ 0) to S_0.

S_0	
$S' ::= \cdot S$	0
$S ::= \cdot aSB$	0
$S ::= \cdot b$	0

Since no more rules can be predicted we continue by scanning the next input symbol, a, and construct the set S_1 with the item ($S ::= a \cdot SB$ 0). We continue by predicting two more items, ($S ::= \cdot aSB$ 1) and ($S ::= \cdot b$ 1), which completes the construction of S_1.

S_0		S_1	
$S' ::= \cdot S$	0	$S ::= a \cdot SB$	0
$S ::= \cdot aSB$	0	$S ::= \cdot aSB$	1
$S ::= \cdot b$	0	$S ::= \cdot b$	1

We construct S_2 by scanning the next input symbol, a, adding the item $(S ::= a \cdot SB\ 1)$ and then predicting two more items $(S ::= \cdot aSB\ 2)$ and $(S ::= \cdot b\ 2)$.

S_0		S_1		S_2	
$S' ::= \cdot S$	0	$S ::= a \cdot SB$	0	$S ::= a \cdot SB$	1
$S ::= \cdot aSB$	0	$S ::= \cdot aSB$	1	$S ::= \cdot aSB$	2
$S ::= \cdot b$	0	$S ::= \cdot b$	1	$S ::= \cdot b$	2

We begin the construction of S_3 by scanning the symbol b and adding the item $(S ::= b\cdot\ 2)$. Since the item has the dot at the end of a rule, we go back to the state indicated by the item's pointer, in this case S_2, collect all items that have a dot before S, move the dot past the symbol and add them to S_3. In this case only one item is added to S_3, $(S ::= aS \cdot B\ 1)$.

We continue by predicting the rule $B ::= \epsilon$ and add the item $(B ::= \cdot\ 3)$. Because the new item added is an ϵ-rule it can be completed immediately. This involves searching over the current state for any items that have the dot before the non-terminal B. In this case we find the item $(S ::= aS \cdot B\ 1)$, which results in the new item $(S ::= aSB\cdot\ 1)$ being added to the current state. We then complete this item by going back to S_1 and finding the item $(S ::= a \cdot SB\ 0)$ which we then add to S_3 as $(S ::= aS \cdot B\ 0)$.

Normally at this point we predict the item $(B ::= \epsilon)$. However, since the item already exists in the current state, we do not add it again to prevent the algorithm from not terminating on left recursive grammars. Instead we perform the any completions that have not already been performed in the current state for the non-terminal B. This results in the addition of the item $(S ::= aSB\cdot\ 0)$ which in turn causes the item $(S' ::= S\cdot\ 0)$ to be added to S_3.

S_0		S_1		S_2		S_3	
$S' ::= \cdot S$	0	$S ::= a \cdot SB$	0	$S ::= a \cdot SB$	1	$S ::= b\cdot$	2
$S ::= \cdot aSB$	0	$S ::= \cdot aSB$	1	$S ::= \cdot aSB$	2	$S ::= aS \cdot B$	1
$S ::= \cdot b$	0	$S ::= \cdot b$	1	$S ::= \cdot b$	2	$B ::= \cdot$	3
						$S ::= aSB\cdot$	1
						$S ::= aS \cdot B$	0
						$S ::= aSB\cdot$	0
						$S' ::= S\cdot$	0

At this point no more items can be created and since we have consumed all the input and the item $(S' ::= S\cdot\ 0)$ is in the final state, the parse terminates in success.

Each item consists of a grammar rule with a dot on its right hand side, a pointer back to the set where we started searching for a derivation using this rule and a lookahead symbol. The dotted rule is that represents the part of the rule that has been used to recognise a portion of the input string, Instead of pre-constructing a DFA for G, the algorithm constructs the state sets on-the-fly. This avoids the need of a stack during a parse. As each input symbol is parsed a new set of items is constructed which represents the rules of the grammar that can be reached after reading the input string.

Earley's algorithm

Earley's algorithm accepts as input a grammar G and a string is $X_1 \dots X_n$. The grammar productions need to be numbered from $1, \dots, d-1$ and the grammar should be augmented, with the augmented rule numbered 0. The $\dashv$ symbol is the end of string terminator instead of \$.

A state is defined as a four tuple $\langle p, j, f, \alpha \rangle$, where p is a production number, j position in the rule, f is the number of the state set S_f that the item has been constructed from and α is the lookahead used.

The state set acts as a queue where every element is added to the end of the set unless it is already a member of the set.

In addition to some data structures used to achieve the required time and space bounds extra searching needs to be done in the case of grammars containing ϵ-rules. When an ϵ reduction, $A ::= \cdot$ is encountered within a state set S_i it is necessary to go through the set S_i and move on any of the items with a dot before the nonterminal A. This must be done cautiously as some of the items that require the move may not have been created yet. It is therefore necessary to check if this reduction is possible when new items are added to the set.

The formal specification of Earley's algorithm shown below is taken from [Ear68]. It is this algorithm that has been implemented and used to compare the performance between the RNGLR and BRNGLR algorithms in Chapter 10.

```
function RECOGNISER($G, X_1 \cdots X_n, k$)
    Let $X_{n+i} = \dashv$ $(1 \leq i \leq k+1)$
    Let $S_i$ be empty $(0 \leq i \leq n+1)$
    Add $\langle 0, 0, 0, \dashv^k \rangle$ to $S_0$

    for $i \leftarrow 0$ step 1 until $n$ do
        Process the states of $S_i$ in order, performing one of the following three
            operations on each state $s = \langle p, j, f, \alpha \rangle$.
        (1) Predictor: If $s$ is nonfinal and $C_{p(j+1)}$ is a non-terminal, then for each
            $q$ such that $C_{p(j+1)} = D_q$, and for each $\beta \in H_k(C_{p(j+2)} \cdots C_{p\bar{p}\alpha})$ add
            $< q, 0, i, \beta >$ to $S_i$.
        (2) Completer: If $s$ is final and $\alpha = X_{i+1} \cdots X_{i+k}$, then for each $\langle q, l, g, \beta \rangle \in$
            $S_f$ (after all states have been added to $S_f$) such that $C_{q(l+1)} = D_p$, add
```

$< q, l+1, g, \beta >$ to S_i.

(3) Scanner: If s is nonfinal and $C_{p(j+1)}$ is terminal, then if $C_{p(j+1)} = X_{i+1}$, add $\langle p, j+1, f, \alpha \rangle$ to S_{i+1}.

If S_{i+1} is empty, return rejection.
If $i = n$ and $S_{i+1} = \{\langle 0, 2, 0, \dashv \rangle\}$, return acceptance.

Earley's algorithm is described as a depth-first top-down parser with bottom-up recognition [GJ90]. In Chapter 2 we discussed how a non-deterministic parse can be performed by using a depth-first search to find a successful derivation, but decided that such an approach is infeasible for practical parsers because of the possible exponential time complexity required. Earley's algorithm restricts the amount of searching that is required by incorporating bottom-up style reductions in his algorithm.

The description given by Earley in his thesis constructs lists of items, however Graham [GH76] describes Earley's algorithm using a recognition matrix, where the item $(A ::= \alpha \cdot \beta, i)$ is in the (i, j)th entry. This is done to allow a comparison between the CYK and Earley algorithms.

Earley's algorithm is often preferred to the CYK algorithm for two main reasons; the grammar does not need to be in any special form and the worst case time and space bounds are not always realised. In fact the algorithm is quadratic on all unambiguous grammars and linear on a large class of grammars which include the bounded state grammars and most LR(k) grammars (excluding some right recursive grammars). However, those LR(k) grammars that are not bounded state can be parsed in linear time if extra lookahead is used. (Note that the lookahead is not needed for the n^3 and n^2 bounds to be realised.)

Earley claims that the recogniser can be easily extended to a parser without affecting the time bounds, but increasing the space bound to n^3, because of the need to store the parse trees. However, the extension presented in [Ear68] has been shown to create spurious derivations for some parses [Tom85].

Earley also says that his algorithm has a large constant coefficient and when compared to the linear techniques does not compare well. This is because the linear parsers usually compile a parser for a given grammar and use it to parse the input without needing to refer to the grammar. A pre-compiled version of the algorithm is presented in [Ear68] but does not work for all grammars. In fact it is shown that determining whether a given grammar is compilable is undecidable and they cannot even be enumerated. Others [AH01, Lee92a, Lee92b, Lee93] have attempted to improve the performance of Earley's algorithm with the use of a pre-compiled table. We only know of one publication [AH01] that reports an implementation and experimental results of this approach.

It is uncertain how the use of lookahead affects the algorithm's efficiency. Earley states that the n^3 and n^2 time bounds can be achieved without lookahead but k symbols of lookahead are required for the algorithm to be linear on LR(k) grammars. It is shown that the number of items in a set can grow indefinitely if lookahead is not used to restrict which items are added by the Completer step. It was later shown by [BPS75] that a better approach is to use the lookahead in the Predictor step instead.

Earley's algorithm has been implemented essentially as written above in PAT and results of the comparison with the BRNGLR algorithm are given in Chapter 10.

8.4 Bernard Lang

Lang developed a general formalism to resolve non-deterministic conflicts in a bottom-up automaton by performing a breadth first search. Tomita's GLR parsing algorithm can be seen as a realisation of Lang's ideas. Unfortunately, due to the complex description of his algorithm, Lang's approach is often overlooked. In this section we discuss the main properties of Lang's algorithm.

Lang's work is an efficient breadth first search approach to dealing with non-determinism in a bottom-up automaton (PDT). Earley's algorithm is a general top-down parsing algorithm and Lang develops a bottom-up equivalent. The algorithm also outputs a context-free grammar whose language is the derivations of a sentence.

Lang uses a PDT and an algorithm $\mathcal{G}$ to calculate all the possible derivations for a sentence d of length n. $\mathcal{G}$ successively builds $n+1$ *item sets* while parsing the input. Each item set $\mathcal{S}_i$ contains the items that are reached after parsing the first i symbols of d. Each item takes the form $((p, A, i), (q, B, j))$ where p and q are state numbers; A and B are non-terminals; i and j are indexes into the input. The algorithm $\mathcal{G}$ uses old items and the PDT transitions to build the new items.

Lang's algorithm is cubic because the PDT only allows one symbol to be popped off the stack in one action. This means that his algorithm does not apply to the natural LR automaton unless either his algorithm or the automaton is modified, or the grammar's rules have a maximum length of two [Sch91]. Modifying the automaton significantly increases its size (the number of actions needed). If Lang's algorithm is modified to allow more than one symbol to be popped off the stack in one action the complexity changes to $O(n^k)$.

8.5 Klaas Sikkel

Klaas Sikkel's book on parsing schemata [Sik97] presents a framework capable of comparing different parsing techniques by abstracting away "algorithmic properties such as data structures or control mechanisms". This framework allows the comparison of different algorithms in a way that facilitates cross fertilisation of ideas across the techniques. This is clearly an important approach, although somewhat more general than the specific goal of this thesis. In particular, parsing schemata are used to examine the similarity of Tomita's and Earley's algorithms.

As part of the analysis undertaken in [Sik97], there is a detailed discussion of Tomita's approach and a description of a parallel bottom-up Tomita parser. The algorithm described is Algorithm 2 and included is a discussion of Kipps' proof that this algorithm is worst case $O(n^{k+1})$. This proof, like Kipps' original, is based on Tomita's Algorithm 1 and ignores the sub-frontiers introduced in Algorithm 2. This is a proof that Algorithm 1 has worst case time complexity $O(n^{k+1})$.

Part III

Implementing Generalised LR parsing

Chapter 9

Some generalised parser generators

Non-deterministic parsing algorithms were used by some of the first compilers [AU73], but the cost of these techniques was high. As a result efficient deterministic parsing algorithms, such as Knuth's LR algorithm, were developed. Although these techniques only parsed a subset of the unambiguous context-free grammars and were relatively difficult to implement, they marked a major shift in the way programming languages were developed; language developers began sacrificing the power of expressiveness for parsing efficiency.

Unfortunately languages with structures that are difficult to generate deterministic parsers for will always exist. One such example is C++, that is mostly deterministic, but includes some problematic structures inherited from C. Stroustrup was convinced to use the LALR(1) parser generator, Yacc, to build a parser for C++, but this proved to be a "bad mistake", forcing him to do a lot of "lexical trickery" to overcome the shortcomings of the deterministic parsing technique [Str94]. Others opted to write C++ parsers by hand instead, but this often resulted in large programs that were difficult to understand. For example, the front end of Edison Design Group's C++ compiler has 404,000 lines of code [MN04].

The result of such problems has seen the development of tools like ANTLR, PRECCx, JavaCC and BtYacc that extend the deterministic algorithms with the use of extra lookahead and backtracking to parse a larger class of grammars. Although it is well known that these approaches can lead to exponential asymptotic time complexities, it is argued that this worst case behaviour is rarely triggered in practice. Unfortunately certain applications like software renovation and reverse engineering often require even more powerful parsing techniques.

Programming languages like Cobol and Fortran were designed before the focus of language designers had shifted to the deterministic parsing techniques. Maintaining this 'legacy' code has become extremely difficult and considerable effort is spent on transforming it into a more manageable format. The enormous amount of code that needs to be modified and the repeatedly changing specifications prohibit a manual implementation of such tools. As a result more and more tools like Bison and the ASF+SDF Meta Environment are implementing fully generalised parsing algorithms.

Another example of the application of generalised parsing techniques can be seen in the development of new programming languages. Microsoft Research utilised a GLR parser during the development of C♯ to carry out a variety of experiments with a "wide range of language-level features that require new syntax and semantics." [HP03].

In this chapter we discuss several tools that extend the recursive descent or standard LR parsing techniques. As the topic of this thesis is GLR parsing we are primarily interested in tools that are extensions of the LR technique. However, we also consider some of the most popular tools that use backtracking to extend recursive descent to non-LL(1) grammars.

9.1 ANTLR

A straightforward approach to parse non-LL(1) grammars is to try every alternate until one succeeds. As long as the grammar does not contain left recursion then this approach will find a derivation if one exists. However, the problem is that such a naïve approach can result in exponential searching costs. Thus the focus of tools that adopt this approach is to try to limit the searching performed. ANTLR (ANother Tool for Language Recognition, formally PCCTS) is a popular parser generator that extends the standard LL technique to parse non-LL(1) grammars with the use of limited backtracking and lookahead [PQ95]. It uses semantic and syntactic predicates that, in particular, allow the user to define which of several successful substring matches should be chosen to continue the parse with.

Semantic predicates, of which there are two: validating, that throw an exception if their conditions fail and disambiguating which try to resolve ambiguities, are declarations that must be met for parsing to continue. Syntactic predicates on the other hand are used to resolve local ambiguity through extended lookahead. Syntactic predicates are essentially a form of backtracking used when non-determinism is encountered during a parse. Semantic actions are not performed until the syntactic predicate has been evaluated. If no syntactic predicates are defined then the parser reverts to using a first-match backtracking technique.

ANTLR generates an LL parser and as a result it cannot handle left recursive grammars. Although left recursion removal is possible [ASU86], the transformation changes the structure and hence potentially the associated semantics of the grammar.

Another top-down parser generator that utilises potentially unlimited lookahead and backtracking is PRECCX (PREttier Compiler-Compiler).

9.2 PRECCx

PRECCx (PREttier Compiler-Compiler (eXtended)) is an LL parser generator that uses a longest match strategy to resolve non-determinism; ideally the rule that matches the largest substring is chosen to continue the parse. In practice, to ensure longest match selection PRECCx relies on the grammar's rules being ordered so that the longest match is encountered first. However, it is not always possible to order the rules in this way. There exist grammars that require different orderings to achieve the longest match on different input strings. This is an illustration of the difficulty in reasoning about generalised versus standard parsing techniques.

9.3 JavaCC

Another tool that extends the recursive descent technique is JavaCC. The Java Compiler Compiler uses an LL(1) algorithm for LL(1) grammars and warns users when a parse table contains conflicts. It can cope with non-deterministic or ambiguous grammars by either setting a global lookahead value to a value greater than 1, or by using a lookahead construct to provide a local hint. Left recursive grammars need to have left recursion removed as is the case with ANTLR and PRECCx.

The remainder of the tools we consider are based on the standard deterministic LR parsing technique.

9.4 BtYacc

Backtracking Yacc is a modified version of the LALR(1) parser generator Berkeley Yacc that supports automatic backtracking and semantic disambiguation to parse non-LALR(1) grammars. It has been developed by Chris Dodd and Vadim Maslov of Siber Systems. The source code is in the public domain and available at `http://www.siber.com/btyacc/`.

When a BtYacc generated parser has to choose between a non-deterministic action, it remembers the current parse point and goes into *trial mode*. This involves parsing the input without executing semantic actions. (There are some special actions that can be used to help disambiguation and hence are executed when in trial mode but these are declared differently.) If the current parse fails, it backtracks to the most recent conflict point and parses another alternate. A trial parse succeeds when all the input has been consumed, or when an action calls the YYVALID construct. The parser then backtracks to the start of the trial parse and follows the newly discovered path executing all semantic actions. This, it is claimed, removes the need for the *lexer feedback hack* to find `typedef` names when parsing C.

Although BtYacc does not require the use of special predicates used by tools such as ANTLR, its approach can lead to the incorrect rejection of valid strings if not used carefully. Furthermore, simple hidden-left recursive grammars can cause BtYacc to fail to terminate on invalid strings. For example the generated parser for Grammar 9.1 fails to terminate when parsing the invalid string aab.

$$\begin{aligned} S &::= BSa \mid a. \\ B &::= \epsilon. \end{aligned} \tag{9.1}$$

BtYacc extends LR parsing using backtracking. The remaining three tools we discuss are based on Tomita's GLR approach that was designed to be more efficient than simple backtracking.

9.5 Bison

An indication that GLR parsing is becoming practically acceptable has come in the inclusion of a generalised parsing algorithm in the widely used parser generator Bison [Egg03]. Although the parsing algorithm implemented is described as GLR, it does not contain any of the sophistication of Tomita's algorithm. In particular, it does not utilise the efficient GSS data structure. Instead it constructs what Tomita describes as a tree structured stack (see Chapter 4). As a consequence of this implementation, Bison cannot be used to parse all context-free grammars. In fact, Bison fails to parse grammars containing hidden-left recursion.

For example, the inefficiency of Bison's GLR mode prevents it from parsing strings of the form b^d, where $d \geq 12$, in Grammar 6.1.

9.6 Elkhound: a GLR parser generator

Elkhound is a GLR parser generator developed at Berkeley. It is directly based on Tomita's approach. Elkhound focuses on the inefficiency of GLR parsers when compared to deterministic techniques such as LALR(1) and uses a hybrid parsing algorithm that chooses when to use GLR or LR on each token processed. It is claimed that this technique produces parsers that are as fast as conventional LALR(1) on deterministic input [MN04].

The underlying parsing algorithm described in [McP02] is the same as the algorithm described by Rekers. The authors attribute the slower execution time of GLR for deterministic grammars to the extra work that needs to be done during reductions. Elkhound improves the performance of its generated GLR parsers by maintaining the deterministic depth of each node in the GSS. The deterministic depth is defined to be the number of edges that can be traversed before reaching a stack node that has an out degree > 1.

For ambiguous grammars it provides the user with an interface that can be used to control the evaluation of semantic values in the case of an ambiguity. Deterministic parsers usually evaluate the semantics associated with a grammar during a parse. A non-deterministic parser requires more sophistication to achieve this since a parse may be ambiguous. Elkhound requires the definition of a special merge function in the grammar that defines the action to take when an ambiguity is encountered. If the ambiguity is to be retained, it is possible to yield the semantic value associated with a reduction only to later discover that the yielded value was ambiguous and hence should have been merged. An approach is presented in [MN04] that avoids this 'merge and yield' problem by ordering the application of reductions.

9.7 Asf+Sdf Meta-Environment

The Asf+Sdf Meta Environment [vdBvDH$^+$01] is a tool, developed at CWI, for automatically generating an Integrated Development Environment (IDE) for domain specific languages from high level specifications. The underlying parsing algorithm is Farshi's algorithm. Asf+Sdf's goal is to provide an environment that can minimise the cost of building such tools, encouraging a high level declarative specification as opposed to a low level operational specification. It has been used successfully to perform large scale transformations on legacy code [Vee03]. This section presents a brief overview of the tool focusing on the parsing algorithm used.

Software renovation involves the transformation or restructuring of existing code into a new specification. Some common uses include "simple global changes in calling conventions, migrations to new language dialects, goto elimination, and control flow restructuring." [DKV99].

One of the problems of performing transformations of legacy code is that there is often no standard grammar for a specific language. In the case of Cobol there is a variety of dialects and many extensions such as embedded CICS, SQL and DB2. Since the deterministic parsing techniques are not compositional these extensions cannot be easily incorporated into the grammar.

Another problem is that the existing grammars of legacy languages are often ambiguous. Since ambiguities can cause multiple parse trees to be generated, the parser is required to select the correct tree.

The front end of compilers and other transformational tools are traditionally made up of a separate lexer and a parser. Certain languages, like PL(1), do not reserve keywords and as a result the lexical analysis, which is traditionally done by a lexer, using regular expressions, is not powerful enough to determine the keywords. As a result the Asf+Sdf Meta Environment uses a *Scannerless* GLR (SGLR) [Vis97] parsing algorithm to exploit of the power of context-free grammars for the lexical syntax as well. The SGLR algorithm is essentially the Rekers/Farshi algorithm where the tokens are individual characters in the input string and one of the four disambiguation filters are implemented.

Asf+Sdf provides four disambiguation filters that can be used to prune trees from the generated SPPF. The primary use of the disambiguation filters in SGLR is to resolve the ambiguities arising from the integration of the lexical and context-free syntax [vdBSVV02]. The implementation of these filters is defined declaratively on a post-parse traversal of the SPPF. However, the parsers performance can be improved if they are incorporated into the parse.

The disambiguation filters used by SGLR are practically useful, but some ambiguities cannot be removed by them. Research is continuing on new techniques for formulating disambiguation filters alongside the parsing algorithm.

Another application that uses the same SGLR parsing algorithm for program transformations is XT [Vis04].

Chapter 10

Experimental investigation

This chapter presents the experimental results based on eight of the generalised parsing algorithms described in this thesis: RNGLR, BRNGLR, RIGLR, Tomita1e, Tomita1e modified, Farshi naïve, Farshi optimised and Earley. Several pathological grammars are used to trigger the algorithms' worst case behaviour and three programming language grammars are used to gauge their performance in practice. All the algorithms' results are given for the recognition time and space required, while the RNGLR and BRNGLR results include statistics on the SPPF construction.

10.1 Overview of chapter

In Chapter 4 we discussed Tomita's GLR parsing algorithms, specifically focusing on his Algorithm 1, that only works for ϵ-free grammars, and Farshi's extension that works for all grammars. We have implemented Farshi's algorithm exactly as it is described in [NF91], and we have also implemented a more efficient version which truncates the searches in the GSS as described in Chapter 4. We refer to the former as Farshi-naïve and the latter as Farshi-opt. We present the results for both algorithms in Section 10.6.

In Chapter 5 we discussed a minor modification, Algorithm 1e, of Tomita's Algorithm 1, which admits grammars with ϵ-rules. On right nullable grammars, Algorithm 1e performs less efficiently with RN parse tables than LR(1) parse tables (although of course sometimes it performs incorrectly on LR(1) tables). We have a modified algorithm (Algorithm 1e mod) which is actually more efficient on RN tables than Algorithm 1e is on the LR(1) table.

We now present experimental data comparing the performance of the different algorithms. RNGLR is compared to Farshi, as these are the two fully general GLR algorithms. However, even with the modifications mentioned above, Farshi's algorithm is not as efficient as Tomita's algorithm, thus the RNGLR algorithm is also compared to Tomita's original approach using Algorithm 1e.

The BRNGLR algorithm is asymptotically faster than the RNGLR algorithm in worst case and data showing this is given. Experiments are also carried out on average cases using Pascal, Cobol and C grammars, to investigate typical as well as worst case performance. The BRNGLR algorithm is also compared to Earley's classic cubic general parsing algorithm and to the cubic RIGLR algorithm, as this was designed to have reduced constants of proportionality by limiting the underlying stack activity.

We also present data comparing the relative efficiency of the GLR algorithms when different types (LR(0), SLR(1), LALR(1) and LR(1)) of parse table are used.

10.2 The Grammar Tool Box and Parser Animation Tool

The Grammar Tool Box (GTB) [JSE04b] is a system for studying language specification and translation. GTB includes a set of built-in functions for creating, modifying and displaying structures associated with parsing.

The Parser Animation Tool (PAT) is an accompanying visualisation tool that has its own implementations of the parsing algorithms discussed in this thesis which run from tables generated by GTB. PAT is written in Java, and GTB in C++ so they necessarily require separate implementations, but this has the added benefit of two authors constructing independent programs from the theoretical treatment. PAT dynamically displays the construction of parse time structures and can also be used in batch mode to generate statistics on parser behaviour. The results presented in this chapter have been generated in this way. A guide to PAT and its use is given in Appendix A.

The complete tool chain conducted for this thesis comprises of `ebnf2bnf`, a tool for converting extended BNF grammars to BNF; GTB which is used to analyse grammars and construct parse tables, and PAT which is used to animate algorithms and generate statistics on parse-time structure size.

10.3 The grammars

Pascal and C typify the top-down and bottom-up approaches to language design. In folklore at least, Pascal is thought of as being designed for LL(1) parsing and C for LALR(1) parsing. In practice, Pascal is reasonably close to LL(1). Our Pascal grammar only has one SLR(1) conflict, arising from the *if-then-else* ambiguity. C is essentially parsable with an LALR(1) parser, but was not initially designed that way. The LALR(1) ANSI-C grammar was only written by Tom Penello in about 1983. Bjarne Stroustrup described at some length the difficulties involved in attempting to build parsers for early versions of C++ using a hybrid of Yacc and a lexer containing much lexical trickery relying on recursive descent techniques [Str94, p.68]. C++'s non-deterministic syntax has clearly stimulated the development of tools such as ANTLR [PQ95] and the GLR mode of Bison [DS04]. Cobol's development was contemporary with that of Algol-60 and thus pre-dates the development of deterministic parsing techniques. The language has a large vocabulary which will challenge any table based parsing method.

For these experiments we have used the grammar for ISO-7185 Pascal extracted from the standard, the grammar for ANSI-C extracted from [KR88] and a grammar for IBM VS-Cobol developed in Amsterdam. This grammar is described in [KL03]. A version of the grammar is available as a hyperlinked browsable HTML file from [Cob]: we used a version prepared for Asf+Sdf from which we extracted the context-free rules. The original grammars are written in EBNF. The `ebnf2bnf` tool was used to generate corresponding BNF grammars. See [JSE04b] for more details of this tool.

To demonstrate the non-cubic behaviour of the RNGLR, Farshi and Tomita 1e algorithms we have used Grammar 6.1 (see page 118), which is discussed in detail in Chapter 6.

10.4 The input strings

For all the input strings used in the experiments we have suppressed lexical level productions and used separate tokenisers to convert source programs into strings of terminals from the main grammar. For instance the Pascal fragment

```
program tree_file(input, output) ;
const MAX_INTSETS = 200; MAX_TREES = 200 ;
function intset_create(var s : intset) : intset_sig ; }
```

is tokenised to

```
program ID ( ID , ID ) ;
const ID = INTEGER ; ID = INTEGER ;
function ID ( var ID : type_ID ) : type_ID ;
```

For Pascal, our source was two versions of a tree viewer program with 4,425 and 4,480 tokens respectively, and a quadratic root calculator with 429 tokens; for ANSI-C a Boolean equation minimiser of 4,291 tokens; and for Cobol, two strings extracted from the test set supplied with the grammar consisting of 56 and 2,196 tokens respectively.

The tokeniser's inability to distinguish between identifiers and type names in C has meant that our C grammar does not distinguish them either. This is the reason for the high number of LR(1) conflicts in the grammars.

The parses for Grammar 6.1 are performed on strings of b's of varying lengths.

10.5 Parse table sizes

This section compares the sizes of the LR(0), SLR(1), LALR(1) and LR(1) parse tables with the corresponding RN tables for our ANSI-C, Pascal and COBOL grammars.

Recall from Chapter 5 that the RNGLR algorithm corrects the problem with Tomita's Algorithm 1e by performing nullable reductions early. This is achieved through the modification of the LR parse table. For a given combination of grammar and LR(0), SLR(1), LALR(1), or LR(1) parse table, the GSS constructed during a parse will be the same size for both RN- and conventional Knuth-style reductions. In general the RN tables will contain far more conflicts, which might be expected to generate more searching during the GSS construction. In Section 10.6 we shall show that, contrary to intuition, the RNGLR algorithm performs better than either Tomita or Farshi's algorithms. In fact it reduces the stack activity compared to the standard LR parsing algorithm for grammars that contain right nullable rules. The LR(0), SLR(1), LALR(1), LR(1), and RN parse tables for each of the programming language grammars are shown in Tables 10.1, 10.2 and 10.3.

ANSI-C	States	Symbols	Conflicts
LR(0)	383	158	391
LR(0) RN	383	158	391
SLR(1)	383	158	88
SLR(1) RN	383	158	88
LALR(1)	383	158	75
LALR(1) RN	383	158	75
LR(1)	1,797	158	421
LR(1) RN	1,797	158	421

Table 10.1: ANSI-C parse table sizes

Pascal	States	Symbols	Conflicts
LR(0)	434	286	768
LR(0) RN	434	286	4,097
SLR(1)	434	286	1
SLR(1) RN	434	286	242
LALR(1)	434	286	1
LALR(1) RN	434	286	233
LR(1)	2,608	286	2
LR(1) RN	2,608	286	1,104

Table 10.2: ISO-Pascal parse table sizes

Cobol	States	Symbols	Conflicts
LR(0)	2,692	1,028	131,506
LR(0) RN	2,692	1,028	167,973
SLR(1)	2,692	1,028	65,913
SLR(1) RN	2,692	1,028	73,003

Table 10.3: IBM VS-Cobol parse table sizes

Cobol requires more than seven times as many states as ANSI-C for LR(0) and SLR(1) tables. In fact GTB's LR(1) table generator ran out of memory when processing Cobol, so we leave those entries empty. GTB builds LALR(1) parse tables by merging LR(1) tables so those entries for Cobol are also blank. We see that this Cobol grammar is highly non-deterministic, reflecting the construction process described in [LV01].

We now consider how the use of different parse tables affects the size of the GSS constructed by our GLR algorithms. Tables 10.4, 10.5 and 10.6 present the number of GSS nodes and edges created during the parse of the C, Pascal and Cobol input strings respectively.

		Tomita/Farshi		RNGLR		BRNGLR	
d	Table	Nodes	Edges	Nodes	Edges	Nodes	Edges
4,291	LR(0)	39,202	39,389	39,202	39,389	41,528	41,748
	SLR(1)	28,479	28,604	28,479	28,604	30,524	30,670
	LALR(1)	28,352	28,476	28,352	28,476	30,397	30,542
	LR(1)	28,323	28,477	28,323	28,477	30,332	30,512

Table 10.4: Comparison between Tomita-style generated GSS sizes for an ANSI-C program

		Tomita/Farshi		RNGLR		BRNGLR	
d	Table	Nodes	Edges	Nodes	Edges	Nodes	Edges
279	LR(0)	1,736	1,750	1,736	1,750	1,943	1,957
	SLR(1)	1,272	1,278	1,272	1,278	1,438	1,444
	LR(1)	1,263	1,266	1,263	1,266	1,425	1,428
4,425	LR(0)	30,607	31,015	30,607	31,015	34,000	34,441
	SLR(1)	21,043	21,258	21,043	21,258	23,592	23,826
	LALR(1)	21,043	21,258	21,043	21,258	23,592	23,826
	LR(1)	21,039	21,135	21,039	21,135	23,540	23,655
4,480	LR(0)	31,336	31,833	31,336	31,833	34,910	35,464
	SLR(1)	21,670	21,974	21,670	21,974	24,360	24,707
	LALR(1)	21,670	21,974	21,670	21,974	24,360	24,707
	LR(1)	21,424	21,569	21,424	21,569	24,000	24,150

Table 10.5: Comparison between Tomita-style generated GSS sizes for Pascal programs

		Tomita/Farshi		RNGLR		BRNGLR	
d	Table	Nodes	Edges	Nodes	Edges	Nodes	Edges
56	LR(0)	513	671	513	671	636	794
	SLR(1)	319	439	319	439	357	110
2,196	LR(0)	19,756	23,002	19,756	23,002	22,897	26,461
	SLR(1)	12,057	13,512	12,057	13,512	13,017	14,517

Table 10.6: Comparison between Tomita-style generated GSS sizes for Cobol programs

The Tomita, Farshi and RNGLR algorithms generate the same structures. The BRNGLR algorithm achieves cubic run times but at the cost of a worst-case constant factor increase in the size of the structures. For any grammar with productions greater than two symbols long the BRNGLR algorithm introduces additional nodes into the GSS. The results show increases of around 10% in the size of the structures.

There is a potential trade off between the amount of nondeterminism in the table and the size of the GSS. Some early reports, [Lan91], [BL89], suggest that LR(1) based GSS's would be much larger than SLR(1) ones because the number of states is so much larger, and in the limit, the number of nodes in a GSS is bounded by the product of the number of table states and the length of the string.

However, this is not necessarily the case. The above tables show LR(1) GSS's that are a little smaller than the corresponding SLR(1) ones. Of course, the LR(1) tables themselves are usually much bigger than SLR(1) tables so the rather small reduction in GSS size might only be justified for very long strings. The Asf+Sdf Meta Environment uses SLR(1) tables.

In the next section we turn our attention to the performance of the GLR algorithms and show how the use of different parse tables affects the performance of the Tomita, Farshi and RNGLR algorithms. Of course, there are pathological cases where the size of an LR(1) GSS is much greater than the corresponding SLR(1) GSS. See [JSE04b].

10.6 The performance of the RNGLR algorithm compared to the Tomita and Farshi algorithms

Tomita's Algorithm 1 was only designed to work on ϵ-free grammars. We have presented a straightforward extension to his algorithm to allow it to handle grammars containing ϵ-rules. Unfortunately this extended algorithm, which we have called Algorithm 1e, may fail to correctly parse grammars with hidden-right recursion. As mentioned in Section 10.1 we have discussed a slightly more efficient version, Algorithm 1e mod, of Algorithm 1e.

In Chapter 5 we discussed the RNGLR algorithm and showed how it extends Tomita's Algorithm 1 to parse all context-free grammars. Our technique is not the first attempt to fully generalise Tomita's algorithm (see Chapter 4). Farshi presented a 'brute-force' algorithm which handles all context-free grammars. However, his approach introduces a lot of extra searching. Here we compare the efficiency of Farshi's algorithm with the RNGLR algorithm. Because Farshi's algorithm is not efficient we also compare RNGLR with Algorithm 1e.

To compare the efficiency of the GSS construction between the different algorithms, the number of edge visits performed during the application of a reduction are counted. An edge visit is only counted when tracing back all possible paths during a reduction. The creation of an edge is not counted as an edge visit. Recall that in the RNGLR and Algorithm 1e mod algorithms the first edge in a reduction path is not visited because of the way pending reductions are stored in the set $\mathcal{R}$.

10.6.1 A non-cubic example

We begin by discussing the results for parses of strings b^d in Grammar 6.1. This grammar has been shown to trigger worst case behaviour for the GLR-style algorithms (see Chapter 6).

d	Algorithm 1e	Algorithm 1e mod	Farshi naïve	Farshi opt	RNGLR
10	1,368	1,091	16,069	3,094	1,091
20	21,828	18,961	513,510	38,254	18,961
30	108,863	98,106	3,845,851	166,989	98,106
40	339,973	313,026	16,046,592	480,799	313,026
50	822,658	768,221	48,629,233	1,101,184	768,221
60	1,694,418	1,598,191	120,387,274	2,179,644	1,598,191
70	3,122,753	2,967,436	259,194,215	3,897,679	2,967,436
80	5,305,163	5,070,456	503,803,556	6,466,789	5,070,456
90	8,469,148	8,131,751	905,648,797	10,128,474	8,131,751
100	12,872,208	12,405,821	1,530,643,438	15,154,234	12,405,821

Table 10.7: Edge visits performed parsing strings in Grammar 6.1.

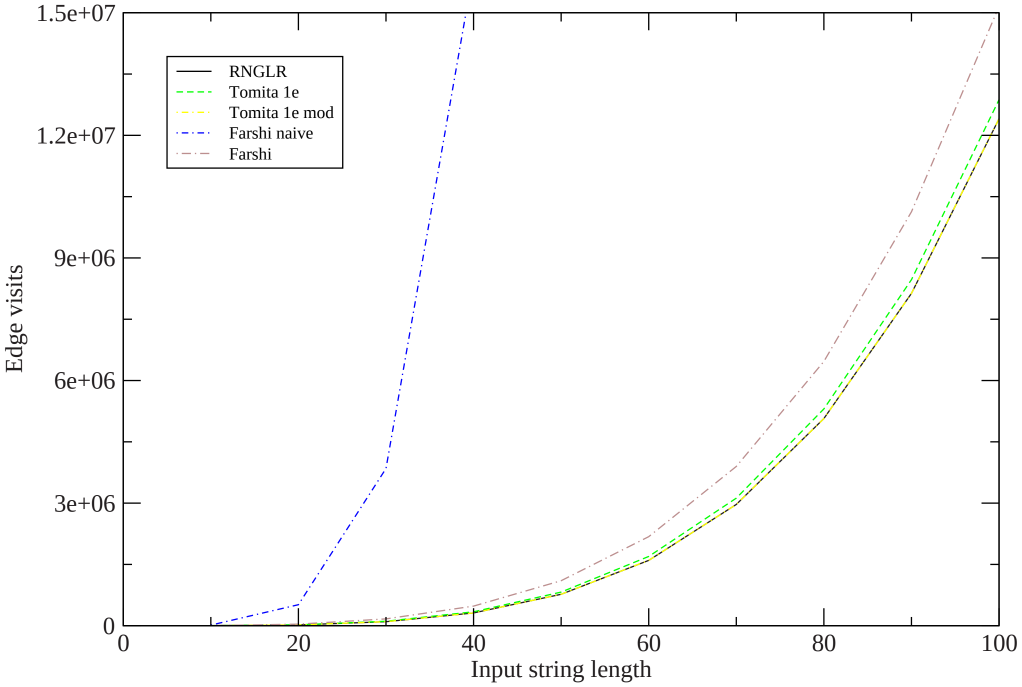


Figure 10.1: Comparison between edge visits done by Tomita-style GLR algorithms for strings in Grammar 6.1.

d	Algorithm 1e		Algorithm 1e mod		Farshi naïve		Farshi opt		RNGLR	
	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
10	38	144	38	144	38	144	38	144	38	144
20	78	589	78	589	78	589	78	589	78	589
30	118	1,334	118	1,334	118	1,334	118	1,334	118	1,334
40	158	2,379	158	2,379	158	2,379	158	2,379	158	2,379
50	198	3,724	198	3,724	198	3,724	198	3,724	198	3,724
60	238	5,369	238	5,369	238	5,254	238	5,369	238	5,369
70	278	7,314	278	7,314	278	7,314	278	7,314	278	7,314
80	318	9,559	318	9,559	318	9,559	318	9,559	318	9,559
90	358	12,104	358	12,104	358	12,104	358	12,104	358	12,104
100	398	14,949	398	14,949	398	14,949	398	14,949	398	14,949

Table 10.8: Comparison between Tomita-style generated GSS sizes for strings in Grammar 6.1.

As we can see, all of the algorithms perform badly as expected, but Farshi's naïve version is much worse than the others. However, the GSS produced by each of the algorithms should be the same and the figures in Table 10.8 support this expectation.

10.6.2 Using different LR tables

In this section we compare the GSS construction costs when different types of LR tables are used.

d	Tables	Algorithm 1e	Algorithm 1e mod	Farshi naïve	Farshi opt	RNGLR
4,291	LR(0)	40,100	5,184	42,707	42,251	5,184
	SLR(1)	28,694	4,502	30,235	29,940	4,052
	LALR(1)	28,567	4,502	30,096	29,801	4,502
	LR(1)	28,461	4,450	30,754	30,484	4,450

Table 10.9: Edge visits performed parsing an ANSI-C Quine McCluskey Boolean minimiser

d	Table	Algorithm 1e	Algorithm 1e mod	Farshi naïve	Farshi opt	RNGLR
279	LR(0)	2,262	857	2,214	2,127	539
	SLR(1)	1,426	519	1,418	1,361	372
	LR(1)	1,401	503	1,350	1,313	364
4,425	LR(0)	39,555	14,620	41,460	38,964	8,556
	SLR(1)	24,008	8,523	25,753	24,100	5,665
	LALR(1)	24,008	8,523	25,753	24,100	5,665
	LR(1)	23,842	8,365	23,305	22,459	5,572
4,480	LR(0)	40,826	15,210	44,472	41,309	8,993
	SLR(1)	25,040	8,978	28,484	26,196	5,976
	LALR(1)	25,040	8,978	28,484	26,196	5,976
	LR(1)	24,201	8,403	24,219	23,289	5,654

Table 10.10: Edge visits performed parsing three ISO-7185 Pascal programs

d	Table	Algorithm 1e	Algorithm 1e mod	Farshi naïve	Farshi opt	RNGLR
56	LR(0)	1,026	579	4,365	3,725	269
	SLR(1)	405	151	2,665	2,316	109
2,196	LR(0)	30,792	15,295	139,187	103,120	10,056
	SLR(1)	13,872	4,478	47,464	38,984	3,581

Table 10.11: Edge visits performed parsing two Cobol programs

GTB and PAT have allowed us to make direct comparisons between the Tomita, Farshi and RNGLR algorithms and three types of LR table. In all cases the LR(1) table resulted in smaller and faster run-time parsers, but the improvement over SLR(1) is not very big, while the increase in the size of the table is significant. Of course, there are pathological cases where the size of an LR(1) GSS is much greater than the corresponding SLR(1) GSS. For example, consider Grammar 10.1 and Grammar 10.2. Both grammars illustrate situations in which all three DFA types are equally bad, but the LR(1) tables result in smaller GSS's being constructed than for the SLR(1) tables.

$$\begin{aligned} S' &::= S \\ S &::= T \mid bTa \\ T &::= aTBB \mid a \\ B &::= b \mid \epsilon \end{aligned} \tag{10.1}$$

$$\begin{aligned} S' &::= S \\ S &::= Ta \\ T &::= aTBB \mid a \\ B &::= b \mid \epsilon \end{aligned} \tag{10.2}$$

d	*Table*	Farshi naïve		Farshi opt		RNGLR	
		Nodes	Edges	Nodes	Edges	Nodes	Edges
20	LR(0)	118	288	118	288	118	288
	SLR(1)	99	269	99	269	99	269
	LR(1)	45	21	29	45	29	45
1000	LR(0)	5,998	504,498	5,998	504,498	5,998	504,498
	SLR(1)	4,999	503,499	4,999	503,499	4,999	503,499
	LR(1)	2,005	1,001	1,009	2,005	1,009	2,005

Table 10.12: GSS sizes for strings of the form b^d in Grammar 10.1.

d	Table	Farshi naïve		Farshi opt		RNGLR	
		Nodes	Edges	Nodes	Edges	Nodes	Edges
20	LR(0)	136	306	136	306	136	306
	SLR(1)	114	266	114	266	114	266
	LR(1)	165	300	165	300	165	300
1000	LR(0)	6,996	505,496	6,996	505,496	6,996	505,496
	SLR(1)	5,994	503,496	5,994	503,496	5,994	503,496
	LR(1)	8,985	505,490	8,985	505,490	8,985	505,490

Table 10.13: GSS sizes for strings of the form b^d in Grammar 10.2.

Table 10.14 shows that there is a significant difference in the run-time cost of a parse for Grammars 10.1 using the LR(1) parse table compared to the SLR(1) and LR(0) parse tables.

d	Table	Farshi naïve	Farshi opt	RNGLR
20	LR(0)	3,252	2,112	190
	SLR(1)	3,233	2,093	190
	LR(1)	402	249	19
1000	LR(0)	334,832,502	168,665,502	499,500
	SLR(1)	334,831,503	168,664,503	499,500
	LR(1)	1,000,002	502,499	999

Table 10.14: Edge visits performed parsing strings of the form b^d in Grammar 10.1.

d	Table	Farshi naïve	Farshi opt	RNGLR
20	LR(0)	3,630	2,319	209
	SLR(1)	2,797	1,828	172
	LR(1)	2,474	1,658	172
1000	LR(0)	335,831,500	169,165,999	500,499
	SLR(1)	333,829,507	168,161,008	498,502
	LR(1)	332,833,504	167,662,508	498,502

Table 10.15: Edge visits performed parsing strings of the form b^d in Grammar 10.2.

10.6.3 The performance of Farshi's algorithms

Although we expected the RNGLR algorithm to carry out significantly fewer edge visits than the either of the Farshi algorithms, it was surprising that the number of edge visits for the naïve version of Farshi were so similar to the optimised version for ANSI-C and Pascal. To get a better understanding of the results, PAT was used to output diagnostics of the algorithm's performance.

The optimised version of Farshi only triggers a saving when a new edge is added to an existing node in the current level of the GSS and the length of the reduction that is reapplied is greater than one. Therefore, if the majority of reductions performed have a length less than two, it is likely that the saving will not be as good as expected. To help reason about this behaviour PAT was used to create histograms of the number of times a reduction of length i was added to the set $\mathcal{R}$ for the ANSI-C and Pascal experiments. The two histograms are shown in Figures 10.2 and 10.3. Their x-axis represents the reduction length and the y-axis the number of times a reduction was added to the set $\mathcal{R}$.

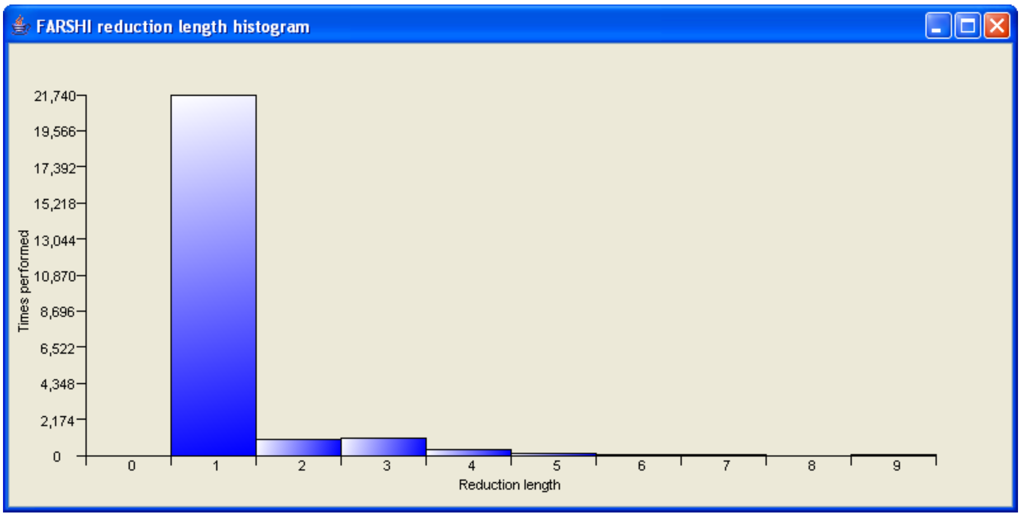


Figure 10.2: Histogram of the number of reductions of length i performed by Farshi's optimised algorithm on the ANSI-C string of length 4,291

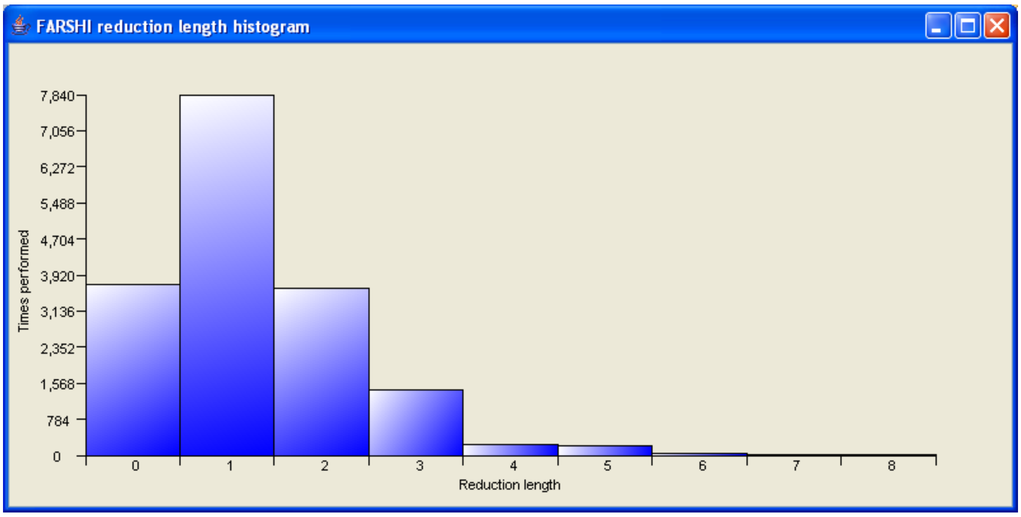


Figure 10.3: Histogram of the number of reductions of length i performed by Farshi's optimised algorithm on the Pascal string of length 4,425

Both histograms show that the majority of reductions performed are of length one which explains why the results are so similar.

The number of edge visits performed parsing strings in Grammar 6.1 differ greatly between the two Farshi algorithms. By counting the number of edge visits contributed by reductions of a certain length, we can see where the optimised version of Farshi is making a saving. We used PAT to generate the histograms in Figures 10.4 and 10.5 that show the number of edge visits performed for reductions of length i in Grammar 6.1. The x-axis represents the different reduction lengths and the y-axis the number of edge visits performed.

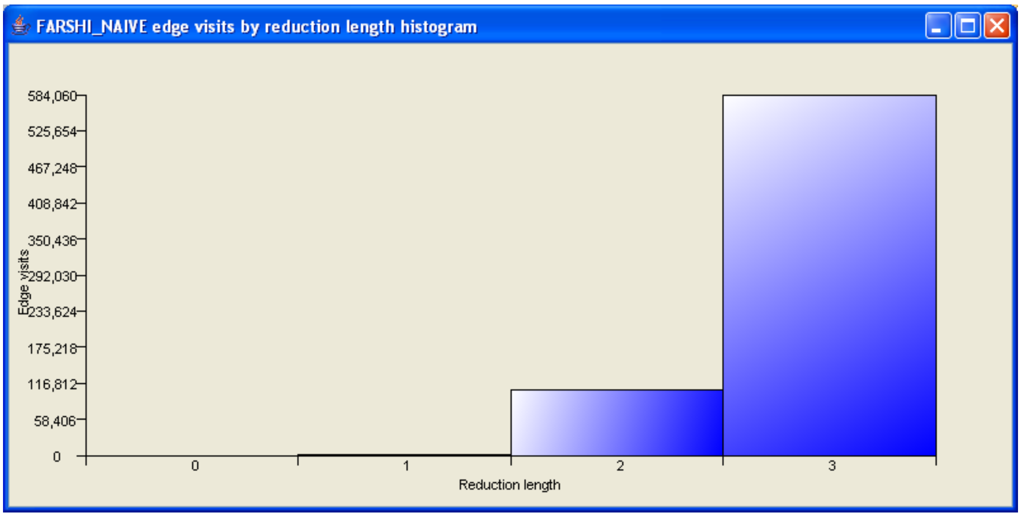


Figure 10.4: Histogram of edge visits for Farshi's naïve algorithm on Grammar 6.1 and a string b^{20}

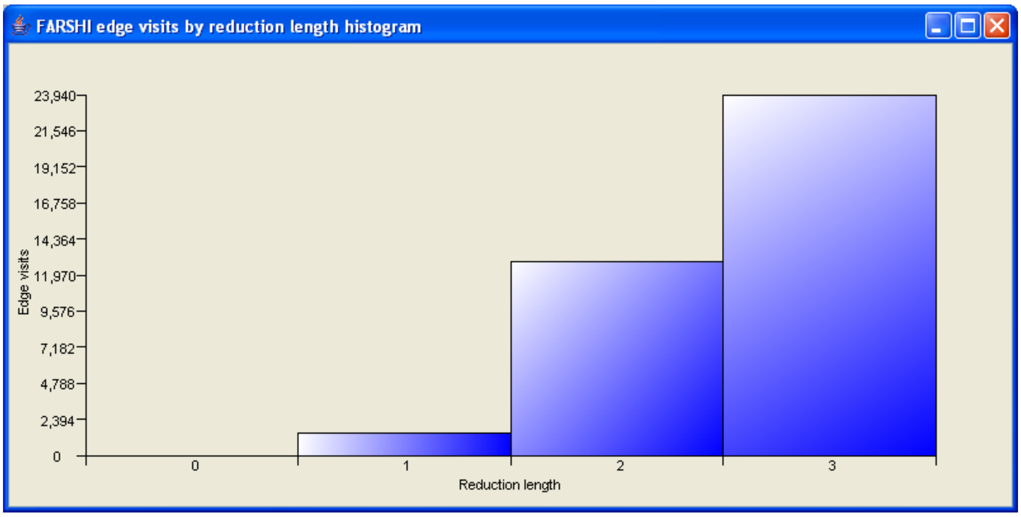


Figure 10.5: Histogram of edge visits for Farshi's optimised algorithm on Grammar 6.1 and a string b^{20}

For Grammar 6.1 there is a difference of 653,620 edge visits because there are few reductions of length one and none of length zero and the other reductions have triggered a significant saving.

In this section we have shown that the RNGLR algorithm is the most efficient of the algorithms compared. However, its worst case complexity is still $O(n^{k+1})$ whereas other algorithms that effectively modify the grammar into 2-form claim cubic worst case complexity. The BRNGLR algorithm presented in Chapter 6 is one such technique. Next we compare the efficiency between the RNGLR and BRNGLR algorithms.

10.7 The performance of the BRNGLR algorithm

This section compares the performance of the BRNGLR and RNGLR parsing algorithms discussed in Chapters 5 and 6 respectively. We present the results of parses for our three programming languages and Grammar 6.1. We begin by discussing the results of the latter experiment which generates worst case behaviour for the BRNGLR algorithm and supra cubic behaviour for the RNGLR algorithm.

10.7.1 GSS construction

The number of edge visits performed by each algorithm are shown in Table 10.16. Recall that, for both algorithms, the creation of an edge is not counted as an edge visit and the first edge in a reduction path is not visited because of the way pending reductions are stored in the set $\mathcal{R}$. As it is sometimes difficult to distinguish between data that has been generated by a cubic, quartic or other polynomial function, the edge visit ratio, which we expect to be linear, is also presented.

d	RNGLR	BRNGLR	Ratio
10	1,091	776	1.41
20	18,961	8,676	2.19
30	98,106	32,676	3.00
40	313,026	81,776	3.83
50	768,221	164,976	4.66
60	1,493,876	291,276	5.13
70	2,800,936	469,676	5.96
80	5,070,456	709,176	7.15
90	8,131,751	1,018,776	7.98
100	12,405,821	1,407,476	8.81
200	199,289,146	11,624,976	17.14

Table 10.16: Edge visits performed parsing strings in Grammar 6.1.

	RNGLR		BRNGLR	
d	Nodes	Edges	Nodes	Edges
10	38	144	11	229
20	78	589	96	1,049
30	118	1,334	146	2,469
40	158	2,379	196	4,489
50	198	3,724	246	7,109
60	238	5,369	296	10,329
70	278	7,314	346	14,149
80	318	9,559	396	18,569
90	358	12,104	446	23,589
100	398	14,949	496	29,209
200	798	59,899	996	118,409

Table 10.17: Comparison between Tomita-style generated GSS sizes for strings in Grammar 6.1.

Because of the regularity of Grammar 6.1 we can compute by hand, the number of edge visits made by the BRNGLR algorithm on b^d. For $d \geq 3$, the number is

$$\frac{3d^3}{2} - \frac{19d^2}{2} + 25d - 24.$$

The experimental results do indeed fit this formula. Table 10.16 only shows a sample of the data gathered. In fact the algorithm was run for all strings b^d for lengths from 1 to 200. This data was exported to Microsoft Excel and used to generate two polynomial trend-lines for the data gathered for both algorithms. Using the RNGLR trend-line we have generated the following formula for the number of RNGLR edge visits, which matches the 201 data points exactly. As expected it is a quartic polynomial.

It is worth noting here that the BRNGLR algorithm implementation parsed b^{1000} in less than 20 minutes, while the 'GLR' version of Bison could not parse b^{20}.

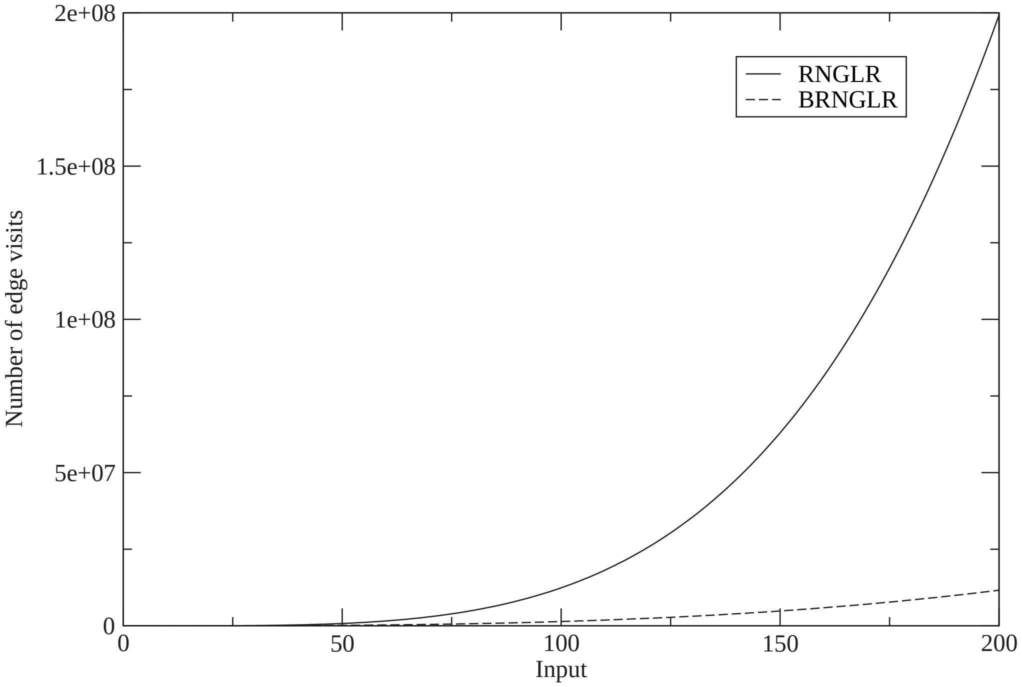


Figure 10.6: Edge visits done by the RNGLR and BRNGLR algorithms for strings b^d in Grammar 6.1

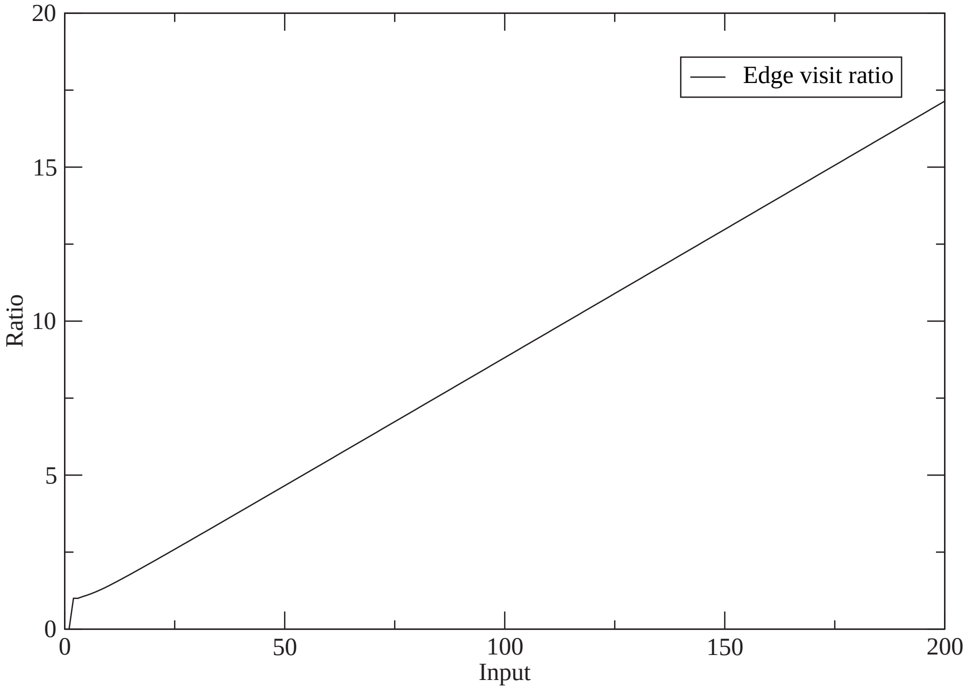


Figure 10.7: Edge visits done by the RNGLR and BRNGLR algorithms for strings b^d in Grammar 6.1

Whilst the BRNGLR algorithm improves the worst case performance, it is of course important that this improvement is not at the expense of average case behaviour. To demonstrate that the BRNGLR algorithm is not less practical than the RNGLR algorithm we have run both algorithms with all three of our programming language grammars. We expect a similar number of edge visits to be performed for both algorithms.

d	Table	RNGLR	BRNGLR
4,291	LR(0)	5,184	5,180
	SLR(1)	4,052	4,498
	LALR(1)	4,502	4,498
	LR(1)	4,450	4,446

Table 10.18: Edge visits performed parsing an ANSI-C Quine McCluskey Boolean minimiser

d	Table	RNGLR	BRNGLR
279	LR(0)	539	539
	SLR(1)	379	379
	LR(1)	364	364
4,425	LR(0)	8,556	8,550
	SLR(1)	5,665	5,663
	LALR(1)	5,665	5,663
	LR(1)	5,572	5,570
4,480	LR(0)	8,993	8,970
	SLR(1)	5,976	5,957
	LALR(1)	5,976	5,957
	LR(1)	5,654	5,654

Table 10.19: Edge visits performed parsing three ISO-7185 Pascal programs

		RNGLR		BRNGLR	
d	Table	Nodes	Edges	Nodes	Edges
279	LR(0)	1,736	1,750	1,943	1,957
	SLR(1)	1,272	1,278	1,438	1,444
	LR(1)	1,263	1,266	1,425	1,428
4,425	LR(0)	30,607	31,015	34,000	34,441
	SLR(1)	21,043	21,258	23,592	23,826
	LALR(1)	21,670	21,974	24,360	24,707
	LR(1)	21,039	21,135	23,540	23,655
4,480	LR(0)	31,336	31,833	34,910	35,464
	SLR(1)	21,670	21,974	24,360	24,707
	LR(1)	21,424	21,569	24,000	24,150

Table 10.20: Comparison between RNGLR and BRNGLR GSS sizes for Pascal programs

d	Tables	RNGLR	BRNGLR
56	LR(0)	269	269
	SLR(1)	109	109
2,196	LR(0)	10,056	9,554
	SLR(1)	3,581	3,487

Table 10.21: Edge visits performed parsing two COBOL programs

		RNGLR		BRNGLR	
d	Table	Nodes	Edges	Nodes	Edges
56	LR(0)	513	671	636	794
	SLR(1)	319	439	357	110
2,196	LR(0)	19,756	23,002	22,897	26,461
	SLR(1)	12,057	13,512	13,017	14,517

Table 10.22: Comparison between RNGLR and BRNGLR GSS sizes for COBOL programs

10.7.2 SPPF construction

As we discussed in Section 6.6, the parser version of the BRNGLR algorithm must retain the cubic order of the underlying recogniser. To illustrate this, data was also collected about the space required for the SPPF's of Grammar 6.1. The number of symbol nodes, additional nodes, packing nodes and edges in the SPPF were recorded. We expect the size of the SPPF generated for Grammar 6.1 by the RNGLR algorithm to be quartic and cubic for the BRNGLR algorithm. To highlight the difference between the two algorithms, the total node ratio, which we expect to be linear, has also been calculated.

b^d	RNGLR			BRNGLR				
d	Symbol nodes	Packing nodes	Edges	Additional nodes	Symbol nodes	Packing nodes	Edges	Total node ratio
10	65	486	1,816	85	65	515	1,615	0.83
20	230	7,296	27,931	460	230	5,325	16,135	1.25
30	495	35,931	139,346	1,135	495	19,435	58,555	1.73
40	860	111,891	437,061	2,110	860	47,845	143,875	2.22
50	1,325	270,676	1,062,076	3,385	1,325	95,555	287,095	2.71
60	1,890	557,786	2,195,391	4,960	1,890	167,565	503,215	3.21
70	2,555	1,028,721	4,058,006	2,555	6,835	268,875	807,235	3.71
80	3,320	1,748,981	6,910,921	3,320	9,010	404,485	1,214,155	4.20
90	4,185	2,794,066	11,05,136	4,185	11,485	579,395	1,738,975	4.70
100	5,150	4,249,476	16,831,655	5,150	14,260	798,605	2,396,695	5.20

Table 10.23: Comparison between RNGLR and BRNGLR generated SPPF's for strings in Grammar 6.1.

The BRNGLR algorithm generates SPPF's an order of magnitude smaller than the RNGLR algorithm for strings in the language of Grammar 6.1. Microsoft Excel was used to generate the following charts of the data.

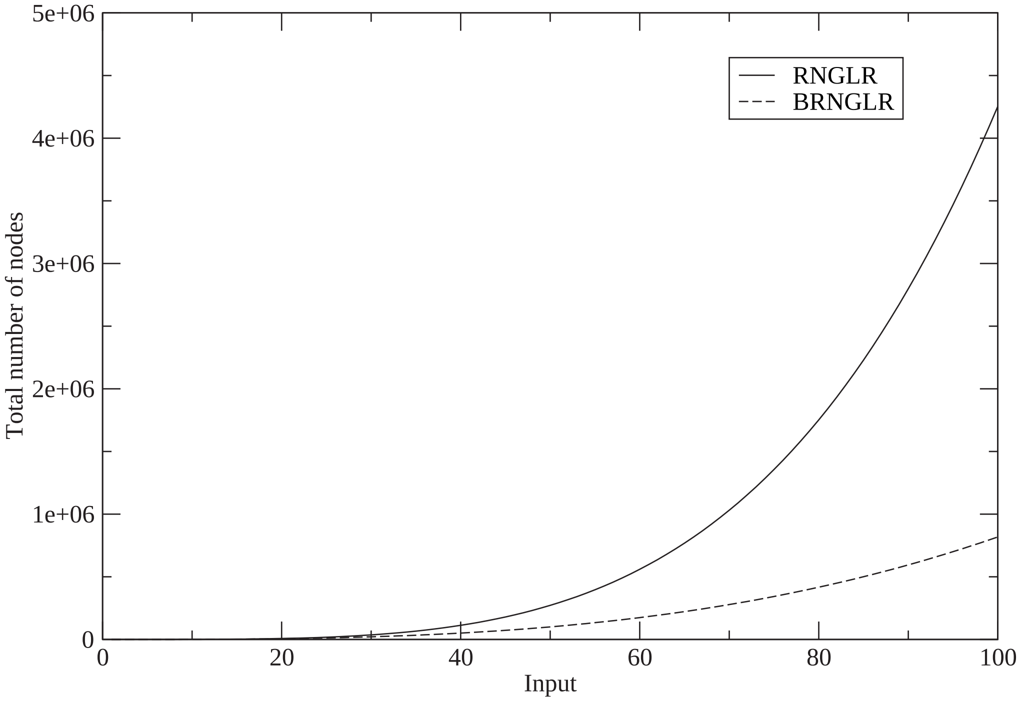


Figure 10.8: Comparison between the sizes of the SPPF's constructed by the RNGLR and BRNGLR algorithms for strings in Grammar 6.1

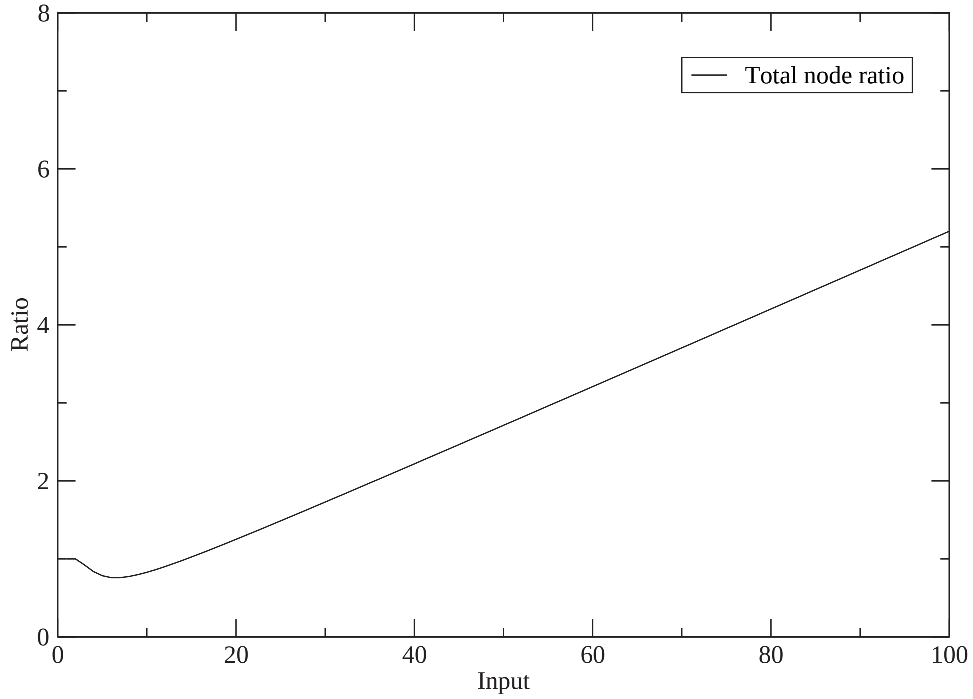


Figure 10.9: Comparison between edge visits done by Tomita-style GLR algorithms for strings in Grammar 6.1

The Excel generated equation for the number of packing nodes created by the RNGLR algorithm is

$$y = 0.0417d^4 + 0.0833d^3 - 0.0417d^2 - 1.0833d - 1.$$

The Microsoft Excel generated equation for the number of packing nodes created by the RNGLR algorithm is

$$y = 0.8333d^3 - 3.5019d^2 + 2.7657d + 3.6201.$$

Because of the irregular way packing nodes are created for $d < 4$, approximation errors occur in the Excel generated charts. By replacing the decimal coefficients with more precise fractions, the correct equation for the number of packing nodes in the RNGLR SPPF can be determined. For $d > 3$ the equation is

$$y = \frac{d^4}{24} + \frac{d^3}{12} - \frac{d^2}{24} - \frac{13d}{12} + 1.$$

Similarly the correct equation for the number of packing nodes in the SPPF generated by the BRNGLR algorithm, for $d > 3$, is

$$y = \frac{5d^3}{6} - \frac{7d^2}{2} + \frac{8d}{3} + 5.$$

To show that the average case performance for the generation of SPPF's is not compromised, SPPF's were also generated for the ANSI-C, Pascal and COBOL programs previously described.

d	Table	RNGLR			BRNGLR			
		Symbol	Packing	Edges	Additional	Symbol	Packing	Edges
		nodes	nodes		nodes	nodes	nodes	
4,291	LR(0)	39,039	364	40,454	2,359	39,039	364	42,800
	SLR(1)	28,301	362	29,033	2,066	28,301	362	31,092
	LALR(1)	28,174	362	28,906	2,066	28,174	362	30,965
	LR(1)	28,093	362	28,761	2,035	28,093	362	30,797

Table 10.24: SPPF statistics for an ANSI-C Quine McCluskey Boolean minimiser

d	Tables	RNGLR			BRNGLR			
		Symbol	Packing	Edges	Additional	Symbol	Packing	Edges
		nodes	nodes		nodes	nodes	nodes	
279	LR(0)	1,372	0	2,100	207	1,372	0	2,307
	SLR(1)	1,090	0	1,445	166	1,090	0	1,611
	LR(1)	1,080	0	1,423	162	1,080	0	1,585
4,425	LR(0)	22,592	6	35,350	3,426	22,592	6	38,770
	SLR(1)	17,190	2	23,325	2,568	17,190	2	25,891
	LALR(1)	17,193	2	23,325	2,568	17,193	2	25,891
	LR(1)	17,065	2	23,050	2,520	17,065	2	25,572
4,480	LR(0)	23,121	21	36,422	3,631	23,121	22	40,031
	SLR(1)	17,624	17	24,158	2,733	17,624	18	26,873
	LALR(1)	17,627	17	24,158	2,733	17,627	18	26,873
	LR(1)	17,264	0	23,304	2,581	17,264	0	25,885

Table 10.25: SPPF statistics for three ISO-7185 Pascal programs

d	Table	RNGLR			BRNGLR			
		Symbol	Packing	Edges	Additional	Symbol	Packing	Edges
		nodes	nodes		nodes	nodes	nodes	
56	LR(0)	556	24	932	123	556	24	1,055
	SLR(1)	422	9	576	38	422	9	614
2,196	LR(0)	16,631	766	27,386	3,459	16,631	818	30,596
	SLR(1)	10,817	306	13,940	1,005	10,817	316	14,898

Table 10.26: SPPF statistics for two COBOL programs

The results presented in this section complement the theoretical analysis of the BRNGLR algorithm. In the next section we compare BRNGLR with Earley's recognition algorithm that is known to also have cubic worst case complexity.

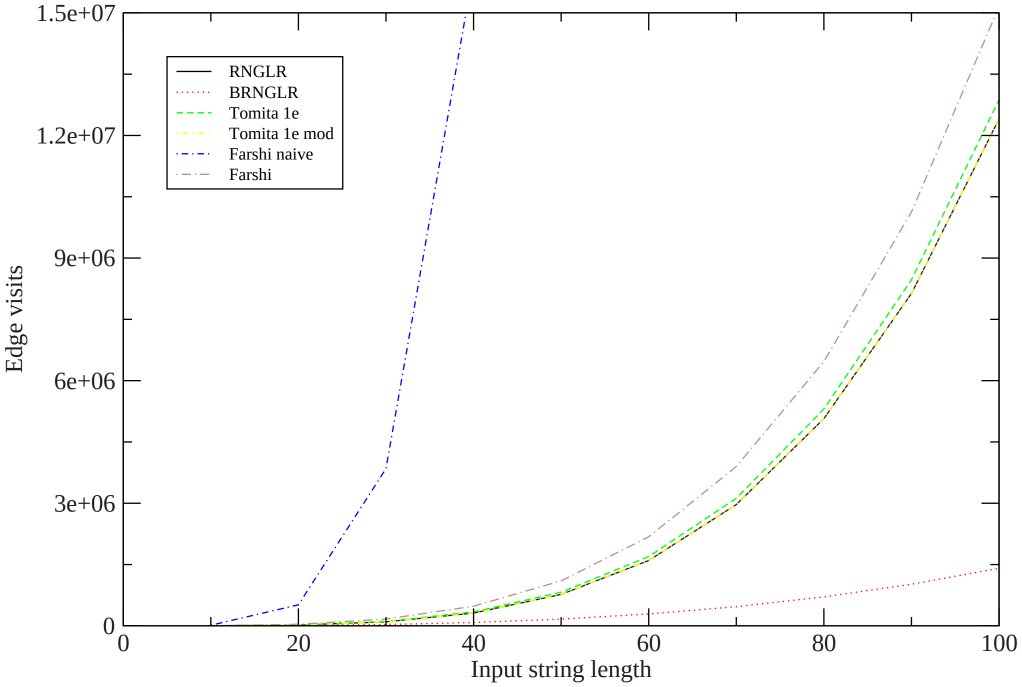


Figure 10.10: Comparison between edge visits done by Tomita-style GLR algorithms for strings in Grammar 6.1

10.8 The performance of Earley algorithm

We discussed Earley's algorithm in Chapter 8. We have implemented his recognition algorithm within PAT and generated statistical data on its performance. We compare the size of the Earley sets with the size of the GSS constructed by the RNGLR and BRNGLR algorithm using an RNLR(1) parse table (except for the COBOL parse which uses an RNSLR(1) table) and the number of symbol comparisons required to construct the Earley sets with the number of edge visits performed by the RNGLR and BRNGLR algorithms.

Earley's algorithm is known to be cubic in the worst case, quadratic for unambiguous grammars and linear on the class of linear bounded grammars [Ear68]. We begin by comparing the performance of the three algorithms during the parse of strings b^d for Grammar 6.1. Recall that this grammar triggers supra cubic behaviour for the RNGLR algorithm and cubic behaviour for the BRNGLR algorithm. Since it is ambiguous we expect Earley's algorithm to display at least quadratic behaviour.

	Earley		RNGLR			BRNGLR		
d	Set size	Symbol comparisons	Nodes	Edges	Edge visits	Nodes	Edges	Edge visits
10	290	1,446	38	144	1,091	11	229	776
20	1,075	12,106	78	589	18,961	96	1,049	8,676
30	2,360	41,966	118	1,334	98,106	146	2,469	32,676
40	4,145	101,026	158	2,379	313,026	196	4,489	81,776
50	6,430	199,286	198	3,724	768,221	246	7,109	164,976
60	9,215	346,746	238	5,369	1,493,876	296	10,329	291,276
70	12,500	553,406	278	7,314	2,800,936	346	14,149	469,676
80	16,285	829,266	318	9,559	5,070,456	396	18,569	709,176
90	20,570	1,184,326	358	12,104	8,131,751	446	23,589	1,018,776
100	25,355	1,628,586	398	14,949	12,405,821	496	29,209	1,407,476
200	100,705	13,177,186	798	59,899	199,289,146	996	118,409	11,624,976

Table 10.27: Earley results for strings in Grammar 6.1

As expected Earley's algorithm performs an order of magnitude fewer symbol comparisons than the RNGLR algorithm's edge visits. It compares well to the BRNGLR algorithm.

Next we present the results of the parses for our three programming languages.

	Earley		RNGLR			BRNGLR		
d	Set size	Symbol comparisons	Nodes	Edges	Edge visits	Nodes	Edges	Edge visits
4,291	283,710	2,994,766	39,202	39,389	4,450	41,528	41,748	4,446

Table 10.28: Earley results for an ANSI-C program

	Earley		RNGLR			BRNGLR		
d	Set size	Symbol comparisons	Nodes	Edges	Edge visits	Nodes	Edges	Edge visits
279	7,648	53,068	1,263	1,266	364	1,425	1,428	364
4,425	133,078	1,012,128	21,039	21,135	5,570	23,540	23,655	5,572
4,480	136,736	1,048,312	21,424	21,569	5,654	24,000	24,150	5,654

Table 10.29: Earley results for Pascal programs

	Earley		RNGLR			BRNGLR		
d	Set size	Symbol comparisons	Nodes	Edges	Edge visits	Nodes	Edges	Edge visits
56	13,275	180,568	319	439	109	357	110	109
2,197	493,795	6,135,997	12,057	13,512	3,581	13,017	14,517	3,487

Table 10.30: Earley results for Cobol programs

Both the RNGLR and the BRNGLR algorithms compare very well to Earley's algorithm for all of the above experiments.

All of the algorithms that we have compared so far have focused on efficiently performing depth-first searches for non-deterministic sentences. In Chapter 7 we presented the RIGLR algorithm which attempts to improve the parsers' efficiency by restricting the amount of stack activity. The next section presents the results collected for the parses of the RIGLR algorithm on our four grammars. We compare the performance of the RIGLR algorithm to the RNGLR and BRNGLR algorithms.

10.9 The performance of the RIGLR algorithm

The asymptotic and typical actual performance of the RIGLR algorithm is demonstrated by comparing it to the RNGLR and BRNGLR algorithms. We use the grammars for ANSI-C, Pascal and Cobol as well as Grammar 6.1. We begin this section by discussing the size of the RIA's and RCA's constructed by GTB and used by PAT for the experiments.

10.9.1 The size of RIA and RCA

Part of the speed up of the RIGLR algorithm over the RNGLR algorithm is obtained by effectively unrolling the grammar. In a way this is essentially a back substitution of alternates for instances of non-terminals on the right hand sides of the grammar rules. Of course, in the case of recursive non-terminals the back substitution process does not terminate, which is why such instances are terminalised before the process begins.

We terminalised and then built the RIA's and RCA's for ANSI-C, Pascal, Cobol and Grammar 6.1 described above. In each case the grammars were terminalised so that all but non-hidden left recursion was removed before constructing the RIA's. Table 10.31 shows the number of terminalised non-terminals, the number of instances of these terminalised non-terminals in the grammar, the number of symbol labelled transitions in RCA(Γ) and the number of reduction labelled transitions in RCA(Γ). (It is a property of RCA(Γ) that the number of states is equal to the number of symbol labelled transitions + the number of terminalised nonterminal instances + 1.)

Grammar	Terminalised non-terminals	Instances of terminalised non-terminals	Symbol transitions	Reduction transitions
ANSI-C	12	42	6,907,535	5,230,022
Pascal	8	11	13,522	11,403
Cobol	19	28	4,666,176	5,174,657
6.1	1	3	5	4

Table 10.31: The size of some RIA's

	States	Symbols
ANSI-C	1,517,419	98
Pascal	288,138	86
Cobol	5,252,943	373

Table 10.32: RCA sizes

The size of the RCA's for ANSI-C and Cobol are impractically large. The potential explosion in automaton size is not of itself surprising: we know it is possible to generate an LR(0) automaton which is exponential in the size of the grammar. The point is that the examples here are not artificial. Some properties of grammars that cause this kind of explosion to occur are discussed in detail in [JSE04a]. The construction of these automata is made difficult because of their size. In particular, the approach of constructing the RCA by first constructing the IRIA's and RIA, as presented in Chapter 7, can be particularly expensive. A more efficient construction approach that involves performing some aspects of the subset construction 'on-the-fly' is presented in [JSE04a].

Ultimately the way to control the explosion in the size of RCA is to introduce more non-terminal terminalisations. In order for the RIGLR algorithm to be correct, it is necessary to terminalise the grammar so that no self embedding exists, but the process is still correct if further terminalisations are applied.

Although this will reduce the size of parser for some grammars, it comes at the cost of introducing more recursive calls when the parser is run and hence the (practical but not asymptotic) run time performance and space requirements of the parser will be increased. Thus there is an engineering tradeoff to be made between the size of the parser and its performance.

10.9.2 Asymptotic behaviour

To measure the run-time performance of the RNGLR and BRNGLR algorithms we count the number of edge visits performed for the application of reductions, in the way described in Section 10.6. For the space required we count the number of state nodes and edges in the GSS as well as the maximum size of the set $\mathcal{R}$. For the RIGLR algorithm we measure the run-time performance by counting the number of RCA edge visits performed during the execution of pop actions and the space by the total number of elements added to the set U_i.

Figure 10.11 and Table 10.33 present the number of edge visits performed by RNGLR, BRNGLR and RIGLR algorithms for parses of the string b^d in Grammar 6.1. The RIGLR and BRNGLR algorithms display cubic time complexity whereas the RNGLR algorithm displays quartic time complexity.

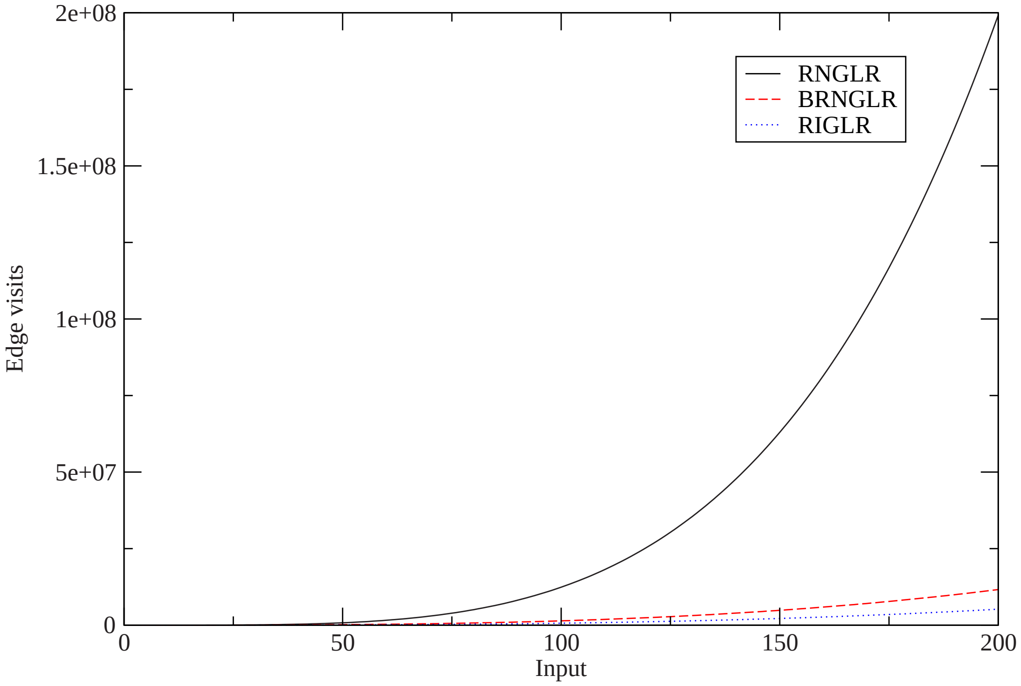


Figure 10.11: Edge visits performed by the RNGLR, BRNGLR and RIGLR algorithms for strings in Grammar 6.1

d	RNGLR	BRNGLR	RIGLR
10	1,091	776	425
20	18,961	8,676	4,255
30	98,106	32,676	15,485
40	313,026	81,776	38,115
50	768,221	164,976	76,145
60	1,598,191	291,276	133,575
70	2,967,436	469,676	214,405
80	5,070,456	709,176	322,635
90	8,131,751	10,18,776	462,265
100	12,405,821	1,407,476	637,295
200	199,289,146	11,624,976	5,214,595

Table 10.33: Edge visits performed parsing strings in Grammar 6.1

	RNGLR				BRNGLR				RIGLR			
d	Nodes	Edges	$\mathcal{R}$	Max $\lvert\mathcal{R}\rvert$	Nodes	Edges	$\mathcal{R}$	Max $\lvert\mathcal{R}\rvert$	Nodes	Edges	U	Max $\lvert U\rvert$
10	38	144	198	36	11	229	283	56	20	165	259	52
20	78	589	893	86	96	1,049	1,353	136	40	725	1,109	112
30	118	1,334	2,088	136	146	2,469	3,223	216	60	1,685	2,559	172
40	158	2,379	3,783	186	196	4,489	5,893	296	80	3,045	4,609	232
50	198	3,724	5,978	236	246	7,109	9,363	376	100	4,805	7,259	292
60	238	5,369	8,673	286	296	10,329	13,633	456	120	6,965	10,509	352
70	278	7,314	11,868	336	346	14,149	18,703	536	140	9,525	14,359	412
80	318	9,559	15,563	386	396	18,569	24,573	616	160	12,485	18,809	472
90	358	12,104	19,758	436	446	23,589	31,243	696	180	15,845	23,859	532
100	398	14,949	24,453	486	496	29,209	38,713	776	200	19,605	29,509	592
200	798	59,899	98,903	986	996	118,409	157,413	1,576	400	79,205	119,009	1,192

Table 10.34: Size of RNGLR and BRNGLR GSS and RIGLR RCA for strings in Grammar 6.1

We now consider the parses performed on the programming language examples.

d	RNGLR	BRNGLR	RIGLR
4,291	5,184	5,180	4,888

Table 10.35: Edge visits performed parsing ANSI-C program

	RNGLR				BRNGLR				RIGLR			
d	Nodes	Edges	$\mathcal{R}$	Max $\lvert\mathcal{R}\rvert$	Nodes	Edges	$\mathcal{R}$	Max $\lvert\mathcal{R}\rvert$	Nodes	Edges	U	Max $\lvert U\rvert$
4,291	28,323	28,477	35,033	3	30,332	30,512	37,392	4	4,009	4,579	59,399	84

Table 10.36: Size of RNGLR and BRNGLR GSS and RIGLR RCA for ANSI-C programs

d	RNGLR	BRNGLR	RIGLR
279	539	539	81
4,425	8,556	8,550	1,523
4,480	8,993	8,970	1,587

Table 10.37: Edge visits performed parsing Pascal programs

	RNGLR				BRNGLR				RIGLR			
d	Nodes	Edges	$\mathcal{R}$	Max $\lvert\mathcal{R}\rvert$	Nodes	Edges	$\mathcal{R}$	Max $\lvert\mathcal{R}\rvert$	Nodes	Edges	U	Max $\lvert U\rvert$
279	1,736	1,750	1,471	5	1,943	1,957	1,678	5	121	121	1,952	25
4,425	30,607	31,015	26,590	5	34,000	34,441	34,441	5	1,830	1,834	34,416	48
4,480	31,336	31,833	27,353	5	34,910	35,464	30,984	5	1,939	1,951	35,463	51

Table 10.38: Size of RNGLR and BRNGLR GSS and RIGLR RCA for Pascal programs

d	RNGLR	BRNGLR	RIGLR
2,197	10,056	9,554	5,759

Table 10.39: Edge visits performed parsing Cobol programs

	RNGLR				BRNGLR				RIGLR			
d	Nodes	Edges	$\mathcal{R}$	Max $\lvert\mathcal{R}\rvert$	Nodes	Edges	$\mathcal{R}$	Max $\lvert\mathcal{R}\rvert$	Nodes	Edges	U	Max $\lvert U\rvert$
2,196	19,756	23,002	19,777	10	22,897	26,461	23,236	10	2,778	5,926	28,802	445

Table 10.40: Size of RNGLR and BRNGLR GSS and RIGLR RCA for Cobol programs

The size of the structures constructed during the parses by the separate algorithms indicate that the RIGLR algorithm achieves an order of magnitude reduction of space compared to the RNGLR and BRNGLR algorithms. The examples also show that the number of edge visits performed by the RIGLR algorithm compare well to the results of the RNGLR and BRNGLR algorithms.

Part IV

Chapter 11

Concluding remarks

11.1 Conclusions

Generalised parsing is an important area of research. The incorporation of a GLR mode in GNU Bison, and the emergence of tools such as the Asf+Sdf Meta-Environment and Stratego/XT highlight the increasing practical importance of generalised parsing techniques. Unfortunately the generalised parsing algorithms used by such tools are still relatively inefficient. Although straightforward optimisations can significantly improve their performance, the literature lacks a comprehensive analysis of the different techniques, which hinders the understanding and improvement of existing approaches and hampers the development of new ideas.

This thesis advances the study of generalised parsing algorithms by providing the following contributions: a comprehensive, comparative analysis of generalised parsing techniques; a new tool, the Parser Animation Tool (PAT), which aids the understanding of the different techniques and can be used to compare their performance; and the BRNGLR algorithm, a new algorithm that displays cubic complexity in the worst case.

The theoretical treatment of the different generalised parsing techniques presented in this thesis is supported by the PAT. The implementation of the algorithms in PAT closely follows their theoretical description. In addition to graphically animating the algorithms' operation PAT collects statistical data which has been used to analyse their performance in Chapter 10. The main results of this work are as follows:

- Both variants of Farshi's algorithms perform poorly when compared to the other techniques.
- The performance of Farshi's algorithm is significantly improved by implementing the straightforward optimisation highlighted in Chapter 4.
- The RNGLR algorithm performs very well in comparison to all the other algorithms for all experiments.

- The BRNGLR algorithm performs an order of magnitude fewer edge visits than the RNGLR algorithm during a parse of a grammar which triggers worst case behaviour for both algorithms.
- The performance of the BRNGLR algorithm also compares very well to the RNGLR algorithm for the programming language parses.
- The performance of the RIGLR algorithm compares well to the BRNGLR and RNGLR algorithms. Additionally there is an order of magnitude reduction in the size of the structures constructed during a parse by the RIGLR algorithm when compared to the RNGLR and BRNGLR algorithms. Unfortunately the size of the RCA's for the programming language grammars is impractically large.

All of the tools which use a GLR parser that were inspected as part of this thesis implement the naïve version of Farshi's algorithm. As the results in this thesis show, a significant improvement in performance can be achieved by making a relatively simple modification to this algorithm.

11.2 Future work

The contributions of this thesis advance the study of generalised parsing, but there are still several areas that can be significantly improved by further research. In the remainder of this chapter we briefly discuss some possibilities for future research in the field.

Resolution of ambiguities

Theoretically speaking it is desirable for a generalised parser to produce all possible derivations of a parse, and the SPPF representation provides a relatively efficient structure to do this with. The problem, however, is that in practice most applications only want their parsers to output one derivation and extracting the desired parse tree from a forest is not usually straightforward. Ambiguities are often difficult to understand and modifying the grammar so as to remove them can result in more ambiguities being introduced and a less intuitive grammar.

New techniques need to be developed and incorporated into generalised parsers that simplify the selection of a single parse tree from a forest. The work done on disambiguation filtering [vdBSVV02] in scannerless GLR parsing is an interesting approach. The incorporation of such techniques in the existing GLR algorithms should not be difficult.

Investigating the effect of the additional GSS nodes created by the BRNGLR algorithm

The theoretical analysis of the BRNGLR algorithm has shown that, in the worst case, it is asymptotically better than the existing GLR algorithms. Furthermore, the results in Chapter 10 indicate that it also performs well in practice. However, it would be interesting to investigate the actual runtime costs that are contributed by the additional nodes created in the GSS. Could the performance of the algorithm be improved in some cases by compromising its worst case complexity and only selectively creating additional nodes?

Using the RIGLR algorithm to improve the performance of scannerless parsers

Scannerless (S)GLR parsers do not use a separate scanner to divide the input string into lexical tokens. Instead they incorporate the lexical analysis phase into the parser. Although this approach has its advantages, one of its main drawbacks is that it can be less efficient; "scanning with a finite automaton has a lower complexity than parsing with a stack" [Vis97, p.36]. Perhaps the efficiency of SGLR parsers can be improved by partially incorporating the techniques developed for the RIGLR algorithm.

Error reporting in GLR parsers

A common complaint against bottom-up parsing techniques in general, is what is considered to be their complicated error reporting. Deterministic bottom-up parsers such as Yacc often refer to parse table actions when a parse error is encountered. Although a considerable amount of research has been done on improving the error reporting for LR parsers, relatively little work has been done for GLR parsers.

Bibliography

[AH99] John Aycock and R. Nigel Horspool. Faster generalised LR parsing. In *Proceedings 8th International Compiler conference*, volume 1575 of *Lecture Notes in Computer Science*, pages 32–46, Amsterdam, March 1999. Springer-Verlag.

[AH01] John Aycock and R. Nigel Horspool. Directly-executable Earley parsing. *Lecture Notes in Computer Science*, 2027:229–243, 2001.

[AH02] John Aycock and R. Nigel Horspool. Practical Earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

[AHJM01] John Aycock, R. Nigel Horspool, Jan Janousek, and Borivoj Melichar. Even faster generalised LR parsing. *Acta Informatica*, 37(9):633–651, 2001.

[aiS] aisee. `http://www.aisee.com`.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[AU73] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation and compiling*, volume I and II. Prentice-Hall, Englewood Cliffs, New Jersey, 1972–1973.

[BL89] Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, Vancouver, British Columbia, Canada, June 1989.

[BP98] Achyutram Bhamidipaty and Todd A. Proebsting. Very fast Yacc-compatible parsers (for very little effort). *Software - Practice and Experience*, 28(2):181–190, February 1998.

[BPS75] M. Bouckaert, A. Pirotte, and M. Snelling. Efficient parsing algorithms for general context-free parsers. *Information sciences*, 8(1):1–26, January 1975.

[BWvW$^+$60] J. W. Backus, J. H. Wegstein, A. van Wijngaarden, M. Woodger, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, and B. Vauquois. Report on the algorithm language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[Cob] `http://www.cs.vu.nl/grammarware/browsable/vs-cobol-ii/`.

[CP82] Keneth Church and Ramesh Patil. Coping with syntactic ambiguity or how to put the block in the box on the table. *American Journal of Computational Linguistics*, 8(3–4):139–149, July–December 1982.

[CS70] John Cocke and Jacob T. Schwartz. Programming languages and their compilers. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[CW87] Don Coppersmith and Shmeul Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 1–6, New York, United States, January 1987. ACM Press.

[CW90] Don Coppersmith and Shmeul Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990.

[DeR69] Franklin L. DeRemer. *Practical translators for LR(k) languages.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1969.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications ACM*, 14(7):435–460, July 1971.

[DKV99] Arie van Deursen, Paul Klint, and Chris Verhoef. Research issues in the renovation of legacy systems. In J. P. Finance, editor, *Fundamental Approaches to Software Engineering, LNCS*, volume 1577, pages 1–21. Springer-Verlag, 1999.

[DS04] Charles Donnelly and Richard M. Stallman. *Bison: The Yacc-Compatible Parser Generator.* Free Software Foundation, 59 Temple Place, Suite 330 Boston, MA 02111-1307 USA, December 2004.

[Ear67] Jay Earley. An n^2 recognizer for context-free grammars. Technical report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, September 1967.

[Ear68] Jay Earley. *An efficient context-free parsing algorithm.* PhD thesis, Computer science department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1968.

[Egg03] Paul Eggert. Bison version 1.875 available. `http://compilers.iecc.com/comparch/article/03-01-005`, January 2003.

[GH76] Susan L. Graham and Michael A. Harrison. Parsing of general context-free languages. *Advances in Computers*, 14:77–185, 1976.

[GHR80] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. An improved context-free recogniser. *ACM transactions on programming languages and systems*, 2(3):415–462, July 80.

[GJ90] Dick Grune and Ceriel Jacobs. *Parsing techniques a practical guide.* Ellis Horwood limited, 1990.

[GR62] S. Ginsburg and H. G. Rice. Two families of languages related to ALGOL. *Journal of the ACM*, 9(3):350–371, 1962.

[HMU01] John E. Hopcroft, Rejeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, second edition, 2001.

[HP03] David R. Hanson and Todd A. Proebsting. A research C# compiler. Technical Report MSR-TR-2003-32, Microsoft Research, 2003.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, first edition, 1979.

[HW90] R. Nigel Horspool and Michael Whitney. Even faster LR parsing. *Software - practice and experience*, 20(6):515–535, June 1990.

[Iro61] Edgar T. Irons. A syntax directed compiler for Algol 60. *Communications of the ACM*, 4(1):51–55, January 1961.

[Joh79] Steven C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[JS02] Adrian Johnstone and Elizabeth Scott. Generalised reduction modified LR parsing for domain specific language prototyping. In *Proc. 35th Annual Hawaii International Conference On System Sciences (HICSS02)*, New Jersey, January 2002.

[JS03] Adrian Johnstone and Elizabeth Scott. Generalised regular parsers. In Gorel Hedin, editor, *Compiler Construction, Proc. 12th Intnl. Conf., CC2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 232–246, Berlin, 2003. Springer-Verlag.

[JSE] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Evaluating GLR parsing algorithms. To appear in Science of Computer Programming.

[JSE04a] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Generalised parsing: some costs. In Evelyn Duesterwald, editor, *Compiler Construction, Proc. 13th Intnl. Conf., CC2004*, volume 2985 of *Lecture Notes in Computer Science*, pages 89–103, Berlin, 2004. Springer-Verlag.

[JSE04b] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. The Grammar Tool Box: a case study comparing GLR parsing algorithms. In Gorel Hedin and Eric Van Wick, editors, *4th Workshop on Language Descriptions, Tools and Applications LDTA2004, also in Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science, 2004.

[JSE04c] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. The Grammar Tool Box GTB and its companion javabased animator tool PAT. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, Barcelona, Spain, April 2004. Elsevier Science.

[Kas65] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachusetts, 1965.

[Kip91] James R. Kipps. GLR parsing in time $o(n^3)$. In Masaru Tomita, editor, *Generalized LR parsing*, chapter 4, pages 42–59. Kluwer Academic Publishers, 1991.

[KL03] Steven Klusener and Ralf Lämmel. Deriving tolerant grammars from a base-line grammar. In *19th International Conference on Software Maintenance (ICSM 2003)*, pages 179–. IEEE Computer Society, September 2003.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[KT69] T. Kasami and K. Torii. A syntax analysis procedure for unambiguous context-free grammars. *Journal of the ACM*, 16(3):423–431, 1969.

[Lan91] M.M. Lankhorst. An empirical comparison of generalized LR tables. In *Workshop on Tomita's algorithm - extensions and applications*, pages 92–98, University of Twente, Computer Science Department, P.O. Box 217, Enschede, The Netherlands, 1991.

[Lee92a] René Leermakers. A recursive ascent Earley parser. *Information Processing Letters*, 41(2):87–91, 1992.

[Lee92b] René Leermakers. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104(2):299–312, 1992.

[Lee93] René Leermakers. *The functional treatment of parsing*. Kluwer academic, 1993.

[Lee02] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM*, 49(1):1–15, 2002.

[LS68] P. M. Lewis, II and R. E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488, 1968.

[LV01] Ralf Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software Practice and Experience*, 31(15):1395–1448, 2001.

[McP02] Scott McPeak. Elkhound: A fast, practical GLR parser generator. Technical Report UCB CSD-2-1214, University of California, Berkeley, Computer science division (EECS) University of California Berkeley, California 94720, December 2002.

[MN04] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Conference on Compiler Constructor (CC04)*, Lecture notes in computer science, Barcelona, Spain, April 2004. Springer-Verlag.

[NF91] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, chapter 5, pages 61–75. Kluwer Academic Publishers, Netherlands, 1991.

[NS96] Mark-Jan Nederhof and Janos J. Sarbo. Increasing the applicability of LR parsing. In Harry Bunt and Masaru Tomita, editors, *Recent advances in parsing technology*, pages 37–57. Kluwer Academic Publishers, Netherlands, 1996.

[Pag70] David Pager. A solution to an open problem by Knuth. *Information and Control*, 17(5):462–473, December 1970.

[PAT05] Parser Animation Tool (PAT). `http://www.cs.rhul.ac.uk/home/giorgios/PAT/`, 2005.

[Pen86] Thomas J. Pennello. Very fast LR parsing. In *SIGPLAN'86: Proceedings of the 1986 SIGPLAN symposium on Compiler Construction*, pages 145–151. ACM Press, 1986.

[Pfa90] Peter Pfahler. Optimizing directly executable LR parsers. In *CC'90: Proceedings of the third international workshop on Compiler Construction*, pages 179–192. Springer-Verlag New York, Inc., 1990.

[PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software - Practice and Experience*, 25(7):789–810, July 1995.

[Rek92] Jan Rekers. *Parser generation for interactive environments.* PhD thesis, University of Amsterdam, 1992.

[Sch91] Yves Schabes. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *Proceedings of the 29th annual meeting on Association for Computational Linguistics*, pages 106–113, Morristown, NJ, USA, 1991. Association for Computational Linguistics.

[She76] B. Sheil. Observations on context-free parsing. *Statistical methods in linguistics*, pages 71–109, 1976.

[Sik97] Klass Sikkel. *Parsing schemata: a framework for specification and analysis of parsing algorithms.* Texts in theoretical computer science, an EATCS series. Springer-Verlag Berlin Heidelberg New York, 1997.

[SJ] Elizabeth Scott and Adrian Johnstone. Generalised bottom-up parsers with reduced stack activity. To appear in The Computer Journal.

[SJ00] Elizabeth Scott and Adrian Johnstone. Tomita style generalised LR parsers. Technical report, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, December 2000.

[SJ03a] Elizabeth Scott and Adrian Johnstone. Reducing non-determinism in reduction modified LR(1) parsers. Technical Report CSD-TR-02-04, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, January 2003.

[SJ03b] Elizabeth Scott and Adrian Johnstone. Table based parsers with reduced stack activity. Technical Report CSD-TR-02-08, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, May 2003.

[SJ04] Elizabeth Scott and Adrian Johnstone. *Compilers and code generation.* Department of Computer Science, Royal Holloway College, University of London, Egham Hill, Surrey, TW20 0EX, 2004.

[SJE03] Elizabeth Scott, Adrian Johnstone, and Giorgios Economopoulos. BRN-table based GLR parsers. Technical Report CSD-TR-03-06, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, July 2003. Journal version submitted, under revision.

[Str94] Bjarne Stroustrup. *The design and evolution of C++.* Addison-Wesley, 1994.

[Tom84] Masaru Tomita. LR parsers for natural languages. In *22nd conference on Association for Computational Linguistics*, pages 354–357, Stanford, California, 1984. Association for Computational Linguistics.

[Tom85] Masaru Tomita. *An efficient context-free parsing algorithm for natural languages and its applications.* PhD thesis, Carnegie Mellon University, December 1985.

[Tom86] Masaru Tomita. *Efficient parsing for natural language: a fast algorithm for practical systems.* Kluwer Academic Publishers, Boston, 1986.

[Tom91] Masaru Tomita, editor. *Generalized LR parsing.* Kluwer academic publishers, 1991.

[Ung68] Stephen H. Unger. A global parser for context-free phrase structure grammars. *Communications of the ACM*, 11(4):240–247, April 1968.

[Val75] L G Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10:308–315, 1975.

[vdBdJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[vdBSVV02] M.G.J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC), LNCS*, volume 2304 of

Lecture Notes in Computer Science, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.

[vdBvDH$^+$01] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction (CC), LNCS*, pages 365–370. Springer, 2001.

[Vee03] Niels Veerman. Revitalizing modifiability of legacy assets. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 19, Washington, DC, USA, 2003. IEEE Computer Society.

[Vis97] Eelco Visser. *Syntax definition for language prototyping.* PhD thesis, University of Amsterdam, 1997.

[Vis04] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer et al, editor, *Domain-specific program generation*, volume 3016 of *Lecture notes in computer science (LNCS)*, pages 216–238. Springer-Verlag, June 2004.

[Woo70] William A. Woods. Context-sensitive parsing. *Commununications of the ACM*, 13(7):437–445, 1970.

[You67] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208, 1967.

Appendix A

Parser Animation Tool user manual

The Parser Animation Tool (PAT) is a Java application that graphically displays and animates the operation of the seven main algorithms discussed in this thesis. This chapter is a manual for the use and installation of PAT. An executable jar file and the entire source of PAT are available from [PAT05].

A.1 Acquisition and deployment of PAT and its associated tools

This section describes how to download and execute PAT and the associated tools.

A.1.1 Java Runtime Environment (JRE)

As PAT is written in Java, an implementation of the Java Virtual Machine (JVM) is required to run PAT. PAT has been developed using Sun Microsystems' Standard Development Kit 1.4.2, but it should be possible to run on any JVM.

A.1.2 Downloading PAT

There are several ways to download PAT depending on the set-up of the target machine. If a compatible version of the JRE with Java Web Start is installed it should be possible to download the application automatically by clicking on the Web Start link. PAT can then be executed from within Web Start or deployed on the local machine. Alternatively there is an option to download a single jar file or a compressed archive of the source files which can then be manually compiled.

The jar file can be executed by either double clicking on its icon or by using the following command `java -jar PAT.jar`.

To compile and run the source files contained in the compressed archive, extract the files onto the local machine and use the command `java PAT.Main` from the top directory of PAT.

By default, Sun's JVM reserves 60Mb of memory to be used by an application it is running. Some parses can create extremely large internal structures, which may cause the JVM to throw an out-of-memory exception. To increase the maximum available memory include the `-Xmx` flag when launching PAT. For example, use the following command to increase the maximum memory to 200Mb: `java -jar -Xmx200m PAT.jar`.

A.1.3 The Grammar Tool Box

The Grammar Tool Box (GTB) is an interpreter for a procedural programming language with facilities for direct manipulation of translator related data structures. A set of built-in functions are used to generate the parse tables used by PAT. Two versions of GTB are used: GTB1.4B generates the LR(0), SLR(1), LR(1) and RN parse tables and GTB2.3.9 is used to build the parse table representation of the RCA used by the RIGLR algorithm.

Both versions can be downloaded from [PAT05].

A.1.4 aiSee

PAT can be used to output a human readable Graph Description Language (GDL) specification of the constructed GSS and SPPF. aiSee is a powerful graph visualisation tool that reads GDL specifications and automatically calculates a customisable graph layout. This is useful when analysing parses that produce very large structures.

A free non-commercial version of aiSee is available on most platforms from [aiS].

A.1.5 Input library

The input library available from [PAT05] contains most of the example grammars, parse tables and input strings used in this thesis. Each grammar appears in a separate directory which in turn contains two sub-directories for the different parse tables:

- `NoNullableAccept` – contains sub-directories for the LR(0), SLR(1) and LR(1) parse tables. Each directory includes the GTB script used to generate the table and all other output of GTB;
- `NullableAccept` – contains sub-directories for the RN-LR(0), RN-SLR(1) and RN-LR(1) parse tables. Each directory includes the GTB script used to generate the table and all other output of GTB.

Below is a full description of the contents of each grammar's directory with a reference to the part of the thesis that the example has been used.

ANSI-C examples

The `ANSIC` directory contains a grammar for ANSI-C, the GTB generated LR(0), SLR(1), LR(1) parse tables, along with their respective GTB scripts, and a tokenised input string of a Boolean equation minimiser (4,291 tokens). These files were used to generate the results presented in Chapter 10.

ISO-7185 Pascal examples

The `ISOPascal` directory contains a grammar for ISO-7185 Pascal, the GTB generated LR(0), SLR(1), LR(1) parse tables, along with their respective GTB scripts, and three tokenised input strings; a quadratic root calculator (429 tokens) and two versions of a tree construction and visualisation program (4,425 and 4,480 tokens).

IBM VS-Cobol examples

The `COBOL` directory contains a grammar for IBM VS-Cobol, the GTB generated LR(0) and SLR(1) parse tables, along with their respective GTB scripts, and a two input strings consisting of 56 tokens and 2,196 tokens respectively.

An example that triggers worst case behaviour in all GLR algorithms

The `Gamma1` directory contain Grammar 6.1, the GTB generated LR(0), SLR(1), LR(1) parse tables, along with their respective GTB scripts, and two tokenised input strings containing one and ten tokens respectively. These files were used to demonstrate the non-cubic behaviour of the GLR algorithms in Chapter 10.

RIGLR examples

The `RIGLR` directory contains the files required by the RIGLR algorithm to parse the languages described above. There are four sub-directories: `ANSIC`, `ISOPascal`, `COBOL` and `Gamma1`. The grammar and string files are all the same as previously described, but the parse tables are generated by GTB2.3.9.

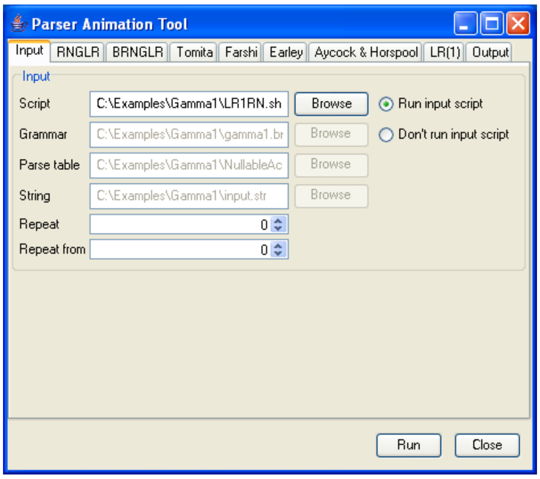
A.2 User interface

In this section we describe the user-interface of PAT and show how to execute an example parse. Figure A.1 shows the first window displayed when PAT is executed (we refer to it as main window). The tabs along the top of the window are used to select the algorithm to parse with and to set the output options.

We begin by describing the format of the different input files.

A.2.1 Specifying the grammar, parse table and string to parse

To perform a parse PAT requires three files as input – a grammar, its associated parse table and a string. Figure A.1 shows the main window of PAT which is used to specify the location of the input files.



Input

Script C:\Examples\Gamma1\LR1RN.sh

Grammar C:\Examples\Gamma1\gamma1.br

Parse table C:\Examples\Gamma1\NullableAc

String C:\Examples\Gamma1\input.str

Repeat 0

Repeat from 0

● Run input script

○ Don't run input script

Figure A.1: PAT's input tab

Grammar format

The format of the grammar input to PAT is standard BNF. Non-terminals are represented by strings of alphanumeric characters while terminals are represented by single quote delimited strings. All the standard BNF operators are used. For a more detailed discussion of BNF see Chapter 2.

PAT does not, as yet, implement automatic augmentation of a grammar. For this reason it is necessary to make the first grammar rule in the file the augmented start rule.

It is essential that the input grammar matches the grammar used to generate the parse tables by GTB. If not, an input error will occur. Grammar files should use the file extension `.bnf`.

Parse table format

The parse tables used by PAT are generated by GTB. The parse table format has been designed to be easy to parse automatically. The file is split up into eight sections: a symbol table; the slots of the NFA; the states of the NFA; the states of the DFA; the reduction table; the start state of the DFA; the accept state of the DFA; and the parse table.

The symbol table is used to allocate unique identifiers to the terminals and non-terminals of the grammar. These identifiers are used throughout the file. The slots and states of the NFA are used to generate the DFA and are mainly used for debugging purposes.

The parse table is represented in a similar way to the parse tables used in this thesis; the rows represent the DFA states and the columns the grammar symbols plus the special end-of-string symbol $. Each row appears on a new line and each column is separated by a tab character. The shift actions are represented by negative integers and the reductions by positive integers which index into the reduction table. If there is a shift action, it always appears first.

It is possible to modify a parse table by hand, but if the language is changed then the appropriate modifications need to be made to the grammar file as well. The parse table files are given the `.tbl` extension.

The parse tables generated by GTB2.3.9 for the use with the RIGLR algorithm need to be modified before they can be used by PAT. The # symbols in the items of the reduction table need to be removed and a new line needs to be added to the end of the file.

Input string format

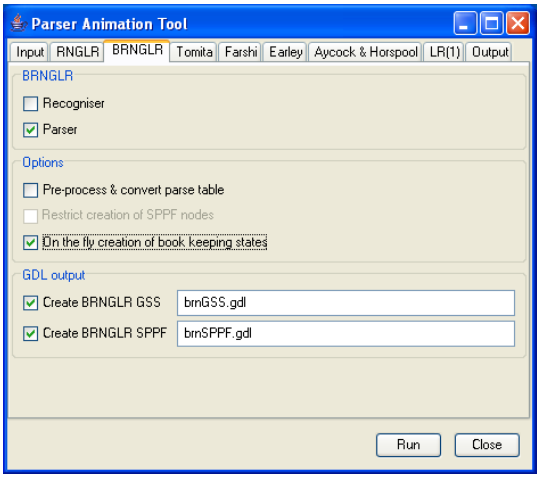
All of PAT's input strings must be pre-tokenised with each token separated by whitespace. The file extension used is `.str`.

Batch file format

The batch file is used to reduce the number of files input to PAT. Instead of specifying the grammar, parse table and input files in separate fields, the batch file can be used to specify the locations of all three. Each path must appear on a separate line and can either be a reference from the directory PAT is executed from or a complete address from the root directory. The file extension used is `.sh`.

A.2.2 Selecting a parsing algorithm

PAT implements seven of the main algorithms discussed in this thesis. Each of the algorithms can be selected from the individual tabs in the main window. Figure A.2 shows the BRNGLR algorithm's tab.



BRNGLR

□ Recogniser

☑ Parser

Options

□ Pre-process & convert parse table

□ Restrict creation of SPPF nodes

☑ On the fly creation of book keeping states

GDL output

☑ Create BRNGLR GSS brnGSS.gdl

☑ Create BRNGLR SPPF brnSPPF.gdl

Figure A.2: The BRNGLR algorithm tab.

The algorithms that can be selected are described below:

RNGLR – the second tab from the left selects the implementations of the RNGLR recogniser and parser. The implementation of both algorithms closely follow the descriptions given on pages 101 and 110 in Chapter 5.

BRNGLR – the third tab selects the implementation of the BRNGLR recognition and parsing algorithms. Either the version that modifies the parse table described on page 125, or the on-the-fly version described on page 127 can be selected. The parser also contains an option to restrict the construction of the SPPF nodes that was used during the development of the BRNGLR algorithm.

Tomita – the fourth tab contains two versions of Tomita's Algorithm 1e recogniser. Tomita 1e is the implementation of the algorithm presented on page 92 and Tomita 1e mod reduces the number of edge visits performed by queueing reductions in the set $\mathcal{R}$ as soon as a new edge is created. Both algorithms are used in the performance comparison presented in Chapter 10.

Farshi – the fifth tab contains the implementation of two versions of Farshi's recognition algorithm. The naïve recogniser implements the algorithm described in Chapter 4. When a new edge is added to an existing node of the GSS, all reduction paths from the current level are re-traversed in case a new reduction path has been created. It is a direct implementation of the description given by Farshi in [NF91]. The other version simply labelled 'Recogniser' is the optimised algorithm described in Section 4.3.2. It stops the search for new reduction paths if a previous level is reached without traversing the newly created edge.

Earley – the sixth tab contains the implementation of Earley's recognition algorithm described on page 190 of Chapter 8.

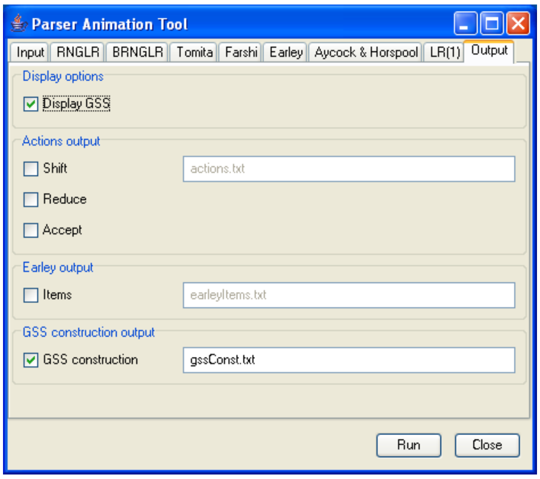
RIGLR – the seventh tab contains the implementation of the RIGLR recognition algorithm presented on page 165 of Chapter 7.

LR – the seventh tab contains an implementation of the standard LR parsing algorithm that uses a GSS instead of a stack during a parse.

The RNGLR, BRNGLR, Tomita and Farshi tabs also contain the options to control the output the GDL specification of the GSS and SPPF constructed. The generated files are written to the location that PAT is executed from.

A.2.3 Output options

PAT can be used to parse strings using one of the generalised algorithms described above. Apart from reporting the success or failure of a parse and generating the GDL of a GSS or SPPF, PAT can also be used to output various traces and to animate the construction of a GSS. The final tab of the main window contains PAT's output options (see Figure A.3).



Display options

☑ Display GSS

Actions output

☐ Shift actions.txt

☐ Reduce

☐ Accept

Earley output

☐ Items earleyItems.txt

GSS construction output

☑ GSS construction gssConst.txt

Figure A.3: PAT's output tab.

By selecting the `Display GSS` option PAT builds a graphical representation of the GSS constructed during a parse and displays it in the Animation window (see Section A.3). However, the extra graphical structures increase the amount of memory consumed by PAT and should only be used when a visualisation of the GSS is needed.

A trace of the parse table actions performed during a parse can be output by selecting one or more of the `Actions output` options. The generated file is written to the directory PAT was executed from.

Since Early's algorithm does not generate a GSS, there is the option of writing the sets of items constructed during a parse to a file. This file is also written to the directory PAT was executed from.

If the `GSS construction` option is selected a file will be created containing a trace of the construction process of the GSS. This includes the creation order of the states and edges in the GSS.

A.3 Animating parses

The animation window is displayed after the run button is pressed. The progress bar shown on the right of the status bar (bottom of the animation window) shows the percentage of levels constructed. Once the parse is complete the execution time is displayed on the left of the status bar and a box appears on the right. A green tick in the box indicates a successful parse and a red cross failure.

If the GSS is to be displayed the toolbar at the top of the screen becomes activated once the parse is completed. Figure A.4 shows the toolbar from the animation window. Each of the buttons have been labelled by numbers to aid the explanation.

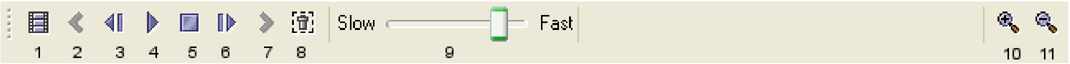


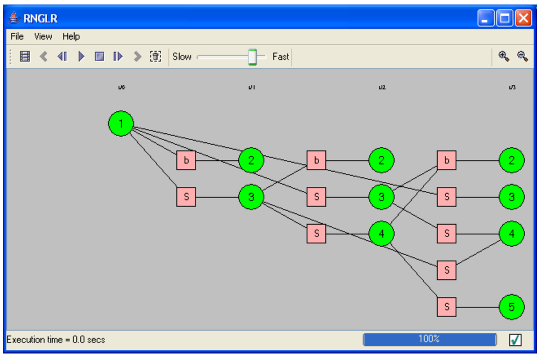
Figure A.4: Animation toolbar.

Buttons 1 & 8	Display and clear the entire GSS from the window
Buttons 2 & 7	Display and remove the next level from the window
Buttons 3 & 6	Display and remove the next node or edge created
Buttons 4 & 5	Start and stop animation of construction of GSS
Button 9	Change speed of animation
Buttons 10 & 11	Zoom in and zoom out

Once the parse is complete pressing button 1 will display the entire GSS as shown in Figure A.5. The GSS is effectively drawn in a grid; the state nodes appear in columns, representing the level they are in, and their vertical position is dependant on their label. The symbol nodes are displayed to make the GSS easier to read. They appear directly to the left of their state node parent.

If the GSS is large it may not be possible to see the labels of the nodes. By clicking once on a state or symbol node its information will be displayed: the label; number of times visited during the construction of the GSS; and node type (state or symbol). An edge can be highlighted by clicking once on the edge. This makes the source and target nodes easier to locate.

The construction of the GSS can be animated with the use of buttons 2 – 7. A node or edge is constructed when it is displayed and the traversal of a reduction path is shown by the path being highlighted.



Execution time = 0.0 secs

Figure A.5: PAT's animation window.

It is possible to open more than one animation window by running one parse after another. This can be useful when analysing and comparing the performance of different algorithms.

A.3.1 GDL output of GSS and SPPF

PAT does not display a graphical representation of an SPPF. However, it is possible to output the specification of an SPPF in a Graph Description Language (GDL) used by other graph visualisation tools such as aiSee or VCG. This GDL output can be selected from the individual algorithm tabs.

In addition to the SPPF's GDL output it is also possible to generate the GDL specification of a GSS. Since these structures can easily become very large, it is useful to use specialised graph visualisation tools to display them.

For example, the SPPF generated by the RNGLR algorithm for the ANSI-C string used in Chapter 10 is shown in Figure A.6.

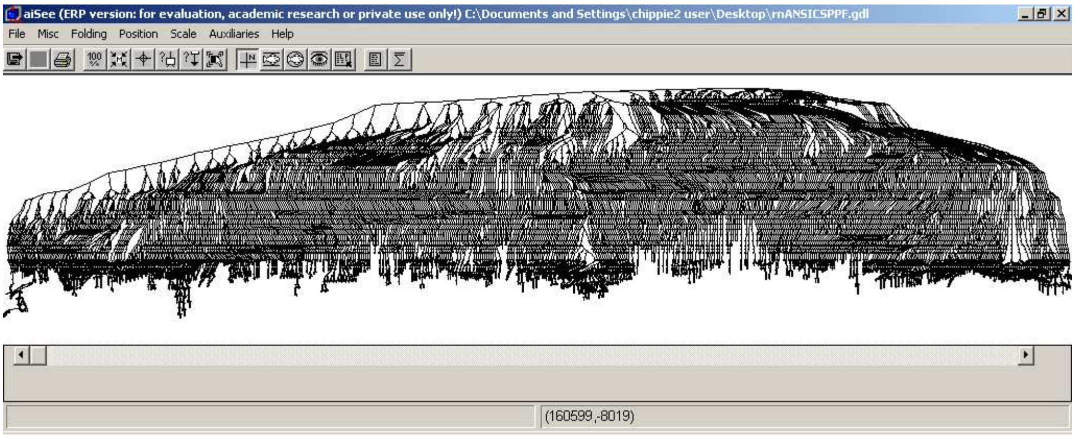

aiSee (ERP version: for evaluation, academic research or private use only!) C:\Documents and Settings\chippie2 user\Desktop\rnANSICSPPF.gdl
File Misc Folding Position Scale Auxiliaries Help
100 %
(160599,-8019)

Figure A.6: The SPPF of the ANSI-C Boolean equation minimiser parsed in Chapter 10.

The files are output in the directory that PAT is executed from and should use the extension `.gdl`.

A.3.2 Histograms

PAT can be used to generate five different histograms from the statistics collected during a parse. The histograms described below can be displayed from the `View` menu of the animation window.

Reduction length histogram	number of times each reduction of length i is performed
Edge visits by reduction length histogram	number of edge visits contributed by reductions of length i
Alternate reduction histogram	number of reductions performed for each alternate in the grammar
Re-applied reductions histogram	number of reductions of length i that are re-applied
Edge visits by re-applied reductions histogram	number of edge visits performed by re-applied reductions

These histograms have been used to analyse the performance of the different algorithms. They proved to be especially useful during the comparison of the two versions of Farshi's algorithm (see Chapter 10). Figure A.7 shows a histogram generated during the parse of a Pascal string.

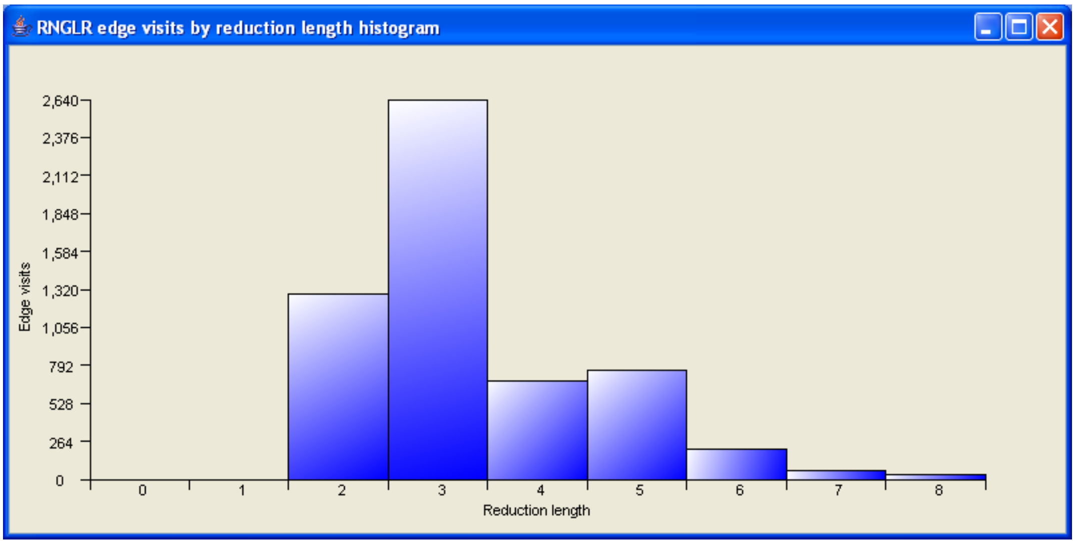


Figure A.7: Histogram showing the number of times reductions of a certain length are performed during the parse of a Pascal string.

A.3.3 Performance statistics

Statistics on the performance of each of the algorithms are automatically collected during a parse. The results can be viewed once a parse is complete by selecting the `Recogniser` or `Parser properties` menu item in the `View` menu.

The data collected during the construction of a GSS for the RNGLR, Tomita and Farshi algorithms are as follows.

Edge visits	number of edge visits performed during the application of a reduction
State nodes	number of state nodes created in the GSS
Symbol nodes	number of symbol nodes created in the GSS
Edges	number of edges created in the GSS
Levels	number of levels created in the GSS
Reductions done	number of reductions performed
Max size of $\mathcal{R}$	the maximum size of the set $\mathcal{R}$ (see pages 59 and 103 for explanation of $\mathcal{R}$)
Symbol node out edges	number of edges leaving symbol nodes
Shift nodes	number of state nodes created as a result of shift actions
Reduce nodes	number of state nodes created as a result of reduce actions
Symbol nodes shared	number of times a symbol node is shared

The BRNGLR algorithm collects the same data, but the state nodes include the number of additional nodes created in the GSS. A separate count of the `Additional nodes` created is also made.

The construction of the SPPF often results in nodes being created that are not included on a path from the root node of the final SPPF. Since the number of nodes and edges created and number of nodes and edges finally used are important both sets of data are collected. The data collected during the construction of the SPPF for the RNGLR parsing algorithm are:

Symbol nodes	number of nodes labelled by grammar symbols in the SPPF used/created
Packing nodes	number of packing nodes in the SPPF used/created
Edges	number of edges in the SPPF used/created
Max size of $\mathcal{N}$	maximum size of the set $\mathcal{N}$ (see page 105 for explanation of $\mathcal{N}$)
Nodes found in $\mathcal{N}$	number of nodes found in the set $\mathcal{N}$

The BRNGLR algorithm collects the same data, but includes an extra field for the number of `Additional nodes` created and used. The number of `Symbol nodes` do not include the additional nodes.

The RIGLR algorithm collects different data from the algorithms described so far. The data collected during a parse are:

Edge visits	number of edge visits performed for pop actions in the RCG
State	number of nodes in the RCG
Edges	number of edges in the RCG
Max size of U	maximum size of the set U
States added to U	total number of nodes added to the set U
States not added to U	number of times an existing node is added to U

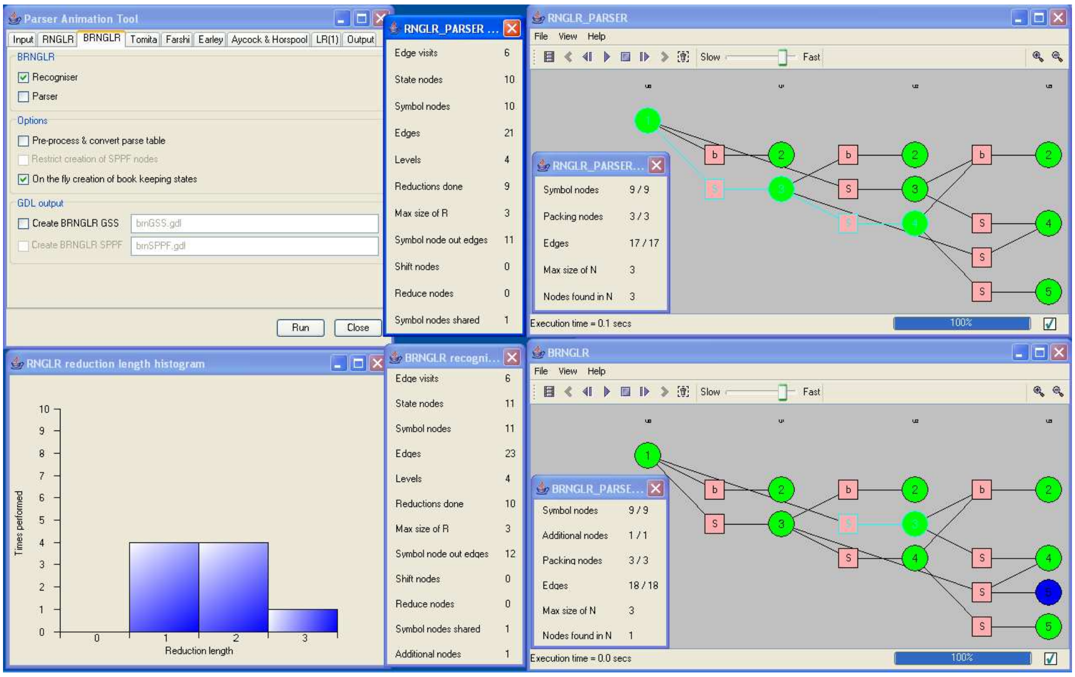
The data collected during a parse of Earley's algorithm are shown below:

Total number of items	number of items created in all sets
Total number of searches performed during reduction predictions	total number of searches done during a reduction for items in a previous state with the dot before the non-terminal reduced on

The results presented in Chapter 10 were collected in this way.

A.3.4 Comparing multiple algorithms in parallel

The original goal of PAT was to provide an environment to demonstrate the internal workings of GLR-style algorithms. However, over time, as more functionality was added, its focus became the generation of experimental data used to compare the performance of generalised parsing algorithms. Despite this shift in focus PAT can still be used to demonstrate the operation of generalised parsers. Figure A.8 shows the animation of the RNGLR and BRNGLR algorithms in parallel.



BRNGLR

☑ Recogniser

☐ Parser

Options

☐ Pre-process & convert parse table

☐ Restrict creation of SPPF nodes

☑ On the fly creation of book keeping states

GDL output

☐ Create BRNGLR GSS brnGSS.gdl

☐ Create BRNGLR SPPF brnSPPF.gdl

RNGLR_PARSER ...

Edge visits	6
State nodes	10
Symbol nodes	10
Edges	21
Levels	4
Reductions done	9
Max size of R	3
Symbol node out edges	11
Shift nodes	0
Reduce nodes	0
Symbol nodes shared	1

BRNGLR recogni...

Edge visits	6
State nodes	11
Symbol nodes	11
Edges	23
Levels	4
Reductions done	10
Max size of R	3
Symbol node out edges	12
Shift nodes	0
Reduce nodes	0
Symbol nodes shared	1
Additional nodes	1

RNGLR_PARSER...

Symbol nodes	9 / 9
Packing nodes	3 / 3
Edges	17 / 17
Max size of N	3
Nodes found in N	3

Execution time = 0.1 secs

BRNGLR_PARSE...

Symbol nodes	9 / 9
Additional nodes	1 / 1
Packing nodes	3 / 3
Edges	18 / 18
Max size of N	3
Nodes found in N	1

Execution time = 0.0 secs

Figure A.8: PAT example