Abstract

gtb is a system for analysing context free grammars. The user provides a collection of BNF rules and uses gtb's programming language, LC, to study grammars built from the rules. In its current stage of development gtb is focused primarily on parsing and the associated grammar data structures. It is possible to ask gtb to produce any of the standard LR DFAs for the grammar, and to create an LR or a GLR parser for the grammar. These parsers can be run as LC methods on any specified input string. gtb also generates other forms of general parsers such as reduction incorporated parsers, Earley parsers and Chart parsers. It produces various forms of output diagnostics, and can be used to compare the different forms of parser and DFA types.

Contents

Chapter 1

Grammars, languages and derivations

1.1 An overview of translation and `gtb`

Computer programs are often written in a so-called ‘high level’ language such as C++ or JAVA. Most human programmers find high level languages easier to use than the ‘low level’ machine oriented languages. However, in order for a machine to execute a program it must be translated from the high level language in which it is written to the native language of that machine. A *compiler* is a program which takes as input a program written in one language and produces as output an equivalent program written in another language.

Although computer languages are designed to be simple to understand and translate, real computer languages still present significant problems. Sometimes, especially with old languages such as FORTRAN and COBOL, the difficulties in translation arise from the imperfect understanding that the early language designers had of the translation process. More modern languages, such as Pascal and Ada, are to a large extent designed to be easy to translate. The discovery that it was possible to design a language which could be translated in linear time (that is the translation time is proportional to the length of the text to be translated) and yet still appear readable to humans was an important result of early work on the theory of programming language syntax.

Computer language translation is traditionally viewed as a process with two main parts: the *front end* conversion of a high level language text written in a language such as C, Pascal or Ada into an *intermediate form*, and the *back end* conversion of the intermediate form into the native language of a computer. This view is useful because it turns out that the challenges encountered in the design of a front end differ fundamentally from the problems posed by back end code generation and separating out the problems makes it easier to think about the overall task.

The language to be translated forms the input to the front end and is called the *source* language. The output of the back end is called the *target* language. In the typical case of a translator that outputs machine code for a particular computer, the target language is usually called the *object code*.

Many of the theoretical issues surrounding front end translation were solved during the 1960s and 1970s, and it is possible to reduce most of the implementation effort for a new front end to a clerical exercise that may itself be turned into a computer program. *Compiler-compilers* are programs that take the description of a programming language, and output the source code of a program that will recognise, and possibly act upon, phrases written in that language. The availability of such tools has fed back into programming language design. It is very hard to use such tools to generate translators for languages such as COBOL, but more recent languages are usually designed in such a way as to facilitate the use of compiler-compilers. The description of the programming language to be input to a compiler-compiler is usually given in a variant of the *generative grammar* formalism which was introduced in the 1950s by Chomsky. The formalism was first applied to the specification of programming languages by John Backus and Peter Naur and in recognition of this the notation used is often called Backus-Naur Form (or BNF). A full discussion of BNF can be found in standard texts such as [ASU86] or [AU72].

However, it is often a complex task to design, for a language, a grammar of the form which allows the standard techniques to be used. Furthermore, in wider areas such as natural language parsing and bioinformatics the languages cannot be selected and thus cannot be designed to be easily parsed. For these reasons there is increasing interest in parsing techniques for grammars which are not of the theoretically tractable forms. `gtb` is a tool that facilitates the study and implementation of algorithms for general context free grammars.

Context free grammars are systems for specifying certain sets of strings of finite alphabets. A set of strings specified by a context free grammar is usually referred to as a (context free) *language*. Not all sets of strings can be specified by a context free grammar, in fact there are sets of strings which cannot be specified using any finite mechanical process. Context free grammars are sufficiently powerful to essentially specify standard programming languages, with the addition of semantic checks to address certain context sensitivities such as type compatibility.

For sets specified by context free grammars, we are usually interested in determining whether or not some given string belongs to the set. The process of determining whether a given string belongs to a given context free language is often referred to as parsing, although strictly speaking parsing also requires the construction of some form of 'derivation' of the string from the grammar.

The parsing problem for context free grammars is known to be less than cubic, that is there are algorithms whose order is less than n^3 which take any context free grammar Γ and any string u of length n and determine whether u belongs to the language defined by Γ. There is no known linear algorithm which performs this task, but there are large sub-classes of context free grammars for which linear parsing algorithms do exist.

This has resulted in extensive investigation of context free grammars, to find better general algorithms, to characterise classes of grammar which can be efficiently parsed, and to support the transformation of grammars into equivalent ones that can be parsed using one of the standard linear time techniques. `gtb` is designed to support all three of these interests. It contains many of the common general parsing techniques, allowing them to be compared and contrasted on different types of grammar. It allows the user to construct different types of automata which form the basis of the standard LR parsers, and it can build many of the structures which support parsing such as FIRST and FOLLOW sets and carry out left recursion detection.

`gtb` can output many of the structures in VCG format. Thus structures such as DFAs, rules trees, grammar non-terminal dependency graphs, parse trees and graph structured stacks to be displayed graphically using the VCG tool [San95].

`gtb` has been designed to support three categories of user: novice grammar users who can use `gtb` to gain an understanding of the principles underlying parsing and grammars, professional programmers who want to develop parsers for real applications, and academic computer scientists who want to understand and extend the theory of grammars and parsing.

This guide is aimed primarily at the first category above, readers who want use `gtb` to extend their understanding of parsing. We shall describe the basic functionality of `gtb`, giving brief discussions of the underlying theory and interspersed with the corresponding `gtb` LC methods and example `gtb` scripts. We begin with an overview of context free grammars and parsing.

1.2 Context free grammars

A context free grammar consists of a set of rewrite rules which are used to generate strings. The strings are generated by replacing instances of the left hand sides of rules with a string from the right hand side of the rule. For example,

$$\begin{array}{lll} S & ::= & S\ +\ S \mid S\ *\ S \mid E \\ E & ::= & a \mid b \end{array}$$

is a set of grammar rules for a grammar ex1 that generates a language of sums and products, for example, `a+b*a+a` or `a`.

Formally, a *context free grammar* consists of a set $\mathbf{N}$ of *non-terminals*, a set $\mathbf{T}$ of *terminals*, and a set $\mathcal{P}$ of *grammar rules* of the form

$$\texttt{A ::= } \alpha_1 \texttt{ | } \alpha_2 \texttt{ | } \ldots \texttt{ | } \alpha_n$$

where A is an element of $\mathbf{N}$ and each α_i is a string of elements from $\mathbf{N}$ and $\mathbf{T}$. One of the non-terminals, S say, is singled out and called the *start symbol.* The strings $\alpha_1, \ldots, \alpha_n$ are called the *alternates* of the rule for A.

In the above grammar, the non-terminals are `S`, `E`, the terminals are `+`,`*`,`a`,`b`, and the start symbol is `S`.

The empty string is denoted by ϵ and it is possible for ϵ to be an alternate of a rule.

How does a grammar specify a language? We *derive* one string from another by replacing a non-terminal with a string from the right hand side of its grammar rule. So if we have a rule

$$A ::= \ldots \mid \gamma \mid \ldots\ .$$

we can replace A by γ. We use the symbol $\Rightarrow$ for a derivation, and we write $\alpha A \beta \Rightarrow \alpha \gamma \beta$.

If μ and τ are strings, we say that τ *can be derived from* μ, and we write $\mu \overset{*}{\Rightarrow} \tau$, if there is a sequence

$$\mu \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \tau.$$

We also write $\mu \overset{+}{\Rightarrow} \tau$ to indicate that the derivation contains at least one step.

For the example above we have

$$\begin{array}{lll}
\texttt{S} & \Rightarrow & \texttt{S + S} \\
 & \Rightarrow & \texttt{E + S} \\
 & \Rightarrow & \texttt{a + S} \\
 & \Rightarrow & \texttt{a + S + S} \\
 & \Rightarrow & \texttt{a + S * S + S} \\
 & \Rightarrow & \texttt{a + E * S + S} \\
 & \Rightarrow & \texttt{a + b * S + S} \\
 & \Rightarrow & \texttt{a + b * E + S} \\
 & \Rightarrow & \texttt{a + b * a + S} \\
 & \Rightarrow & \texttt{a + b * a + E} \\
 & \Rightarrow & \texttt{a + b * a + b}
\end{array}$$

and so $\texttt{S} \overset{*}{\Rightarrow} \texttt{a} + \texttt{b} * \texttt{a} + \texttt{b}$.

The *language specified by a grammar* is the set of strings of terminals which can be derived from its start symbol. We say that $u \in \mathbf{T}^*$ is a *sentence* if $S \overset{*}{\Rightarrow} u$. (Here $\mathbf{T}^*$ denotes the set of strings of elements of $\mathbf{T}$ and includes the empty string ϵ, so if $\mathbf{T}=\{a, b, +\}$ then
$\mathbf{T}^* = \{\epsilon, a, b, +, aa, ab, a+, ba, bb, b+, +a, +b, ++, aaa, aab, \ldots\}$.)

The language generated by the grammar above is the set of all sums and products of a's and b's.

The above derivation is a *left-most* derivation, at each step the left-most non-terminal in the string is replaced. There is a corresponding *right-most* derivation in which the right-most non-terminal is replaced at each step.

$$\begin{array}{lll}
\texttt{S} & \Rightarrow & \texttt{S + S} \\
 & \Rightarrow & \texttt{S + S + S} \\
 & \Rightarrow & \texttt{S + S + E} \\
 & \Rightarrow & \texttt{S + S + b} \\
 & \Rightarrow & \texttt{S + S * S + b} \\
 & \Rightarrow & \texttt{S + S * E + b} \\
 & \Rightarrow & \texttt{S + S * a + b} \\
 & \Rightarrow & \texttt{S + E * a + b} \\
 & \Rightarrow & \texttt{S + b * a + b} \\
 & \Rightarrow & \texttt{E + b * a + b} \\
 & \Rightarrow & \texttt{a + b * a + b}
\end{array}$$

A *sentential form* is a string α, which may include both terminals and non-terminals, such that $S\overset{*}{\Rightarrow}\alpha$.

1.3 Formal grammars and gtb

The input to gtb consists of a set of grammar rules and an LC script of methods to be executed. The rules have to be in a specific format: the terminal symbols are enclosed in single quotes, the left hand sides of the rules are assumed to be the non-terminals, and each grammar rule is terminated by a full stop. There should only be one rule for each non-terminal. The empty string is denoted by # in a gtb grammar.

The following is a gtb input file which specifies the rules for the small grammar, ex1, above.

```
(* ex1 *)

S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .

(* the LC instructions are enclosed in parantheses *)
(
ex1_grammar := grammar[S]

generate[ex1_grammar 10 left sentences]
generate[ex1_grammar 15 right sentential_forms]
)
```

Comments can be included in the file using brackets of the form (* *). The method

```
my_grammar:=grammar[S]
```

causes gtb to make a grammar using the specified rules and start symbol S. The grammar construct is referred to using the variable name my_grammar.

The method call

```
generate[ex1_grammar 10 left sentences]
```

gets gtb to generate 10 strings in the language of the grammar using left-most derivations. The method call

```
generate[ex1_grammar 15 right sentential_forms]
```

gets gtb to generate 15 strings using right-most derivations.

If we run the above script through gtb, using the command

```
gtb ex1.gtb
```

we get the following output.

```
Generated sentences using leftmost derivation

     1: a
     2: b
     3: a + a
     4: a + b
     5: b + a
     6: b + b
     7: a * a
     8: a * b
     9: b * a
    10: b * b

Generated sentential forms using rightmost derivation

     1: S
     2: S '+' S
     3: S '*' S
     4: E
     5: S '+' S '+' S
     6: S '+' S '*' S
     7: S '+' E
     8: S '*' S '+' S
     9: S '*' S '*' S
    10: S '*' E
    11: 'a'
    12: 'b'
    13: S '+' S '+' S '+' S
    14: S '+' S '+' S '*' S
    15: S '+' S '+' E
```

1.4 FIRST and FOLLOW sets

There are several sets associated with the grammar symbols which are used by parsing algorithms to assist in the construction of derivations. In this section we shall discuss two of these sets, the FIRST and FOLLOW sets. We begin with a brief discussion of the two most common approaches to parsing, then describe the roles of the FIRST and FOLLOW sets in these techniques. We then discuss these sets in relation to `gtb`.

1.4.1 Top-down and bottom-up derivation

We often classify derivation techniques as either top-down or bottom up. In a top down technique we begin with the start non-terminal and choose alternates to replace non-terminals until the required string is generated. In a bottom-up technique we read the required string (from the left) until the alternate of a rule is found, and this alternate is then replaced by non-terminal on the left hand side of the corresponding rule. The goal of the bottom-up approach is to end up with a string consisting of just the start symbol.

For example, if we take the grammar above and read the string $a * b + a$ from the left we read a which is the right hand side of the rule $E ::= a$ so we replace a with E, to get the string $E * b + a$. Reading this string we replace E with S, giving $S * b + a$. From this string we read S, $*$ and then b. We can then replace b, which is the right hand side of a rule $E ::= b$, with E, giving $S * E + a$. Next we read S, $*$ and then E, and then replace E with S to give $S * S + a$. Carrying on in this way we generate the strings $S + a$, $S + E$, $S + S$ and finally S.

The top-down and bottom-up approaches rely on pre-computed sets, the FIRST and FOLLOW sets, to help determine which alternate or rule to use.

1.4.2 FIRST sets

If we intend to use the rule $S ::= \alpha$ as the first step in the generation (or top-down derivation) of a string $a_1 \dots a_n$ then it is clear that it must be possible to derive a string beginning with a_1 from α. (If this is not the case then we should use a different alternate of S, if there is one.) In general, when deciding which alternate to use at a step in some derivation it is useful to know whether that alternate can derive a string which begins with a given terminal. The set of terminals which can begin a string derivable from some given string β is called the FIRST set of β.

Formally we define

$$\text{FIRST}_{\mathbf{T}}(\alpha) = \{t \in \mathbf{T} \mid \alpha \overset{*}{\Rightarrow} t\beta, \text{ for some } \beta\}$$

If $\alpha \overset{*}{\Rightarrow} \epsilon$ then we also add ϵ to the FIRST set. So we have

$$\text{FIRST}(\alpha) = \begin{cases} \text{FIRST}_{\mathbf{T}}(\alpha) \cup \{\epsilon\} & \text{if } \alpha \overset{*}{\Rightarrow} \epsilon \\ \text{FIRST}_{\mathbf{T}}(\alpha) & \text{otherwise.} \end{cases}$$

For example, for the above grammar ex1 we have

$$\text{FIRST}(E) = \{a, b\} = \text{FIRST}(S + S)$$

1.4.3 FOLLOW sets

In order to decide whether or not to replace, in a bottom-up derivation, the right hand side of a rule by its left hand side we may use the FOLLOW sets (for more detail see Section 2.3). The FOLLOW set of a non-terminal or a terminal x is the set of terminals which can appear directly to the right of x in a sentential form.

Formally we define

$$\text{FOLLOW}_{\mathbf{T}}(x) = \{t \in \mathbf{T} \mid S \overset{*}{\Rightarrow} \alpha x t \beta, \text{ for some } \alpha, \beta\}$$

If $S \overset{*}{\Rightarrow} \alpha x$ then we also add the special end-of-string symbol, \$, to the FOLLOW set. So we have

$$\text{FOLLOW}(x) = \begin{cases} \text{FOLLOW}_{\mathbf{T}}(x) \cup \{\$\} & \text{if } S \overset{*}{\Rightarrow} \alpha x \\ \text{FOLLOW}_{\mathbf{T}}(x) & \text{otherwise.} \end{cases}$$

For example, for the above grammar we have

$$\text{FOLLOW}(S) = \{\$, +, *\} = \text{FOLLOW}(E)$$

1.4.4 FIRST and FOLLOW sets in `gtb`

When it reads a call to the method `grammar[S]`, `gtb` builds a grammar from the given rules assuming the start symbol S. As part of this process `gtb` constructs various data structures associated with the grammar, including the FIRST and FOLLOW sets for each terminal and non-terminal.

Because `gtb` uses the FIRST and FOLLOW sets in various ways in other parts of its functionality, the sets it constructs differ slightly from the formal definition above in that they can also include non-terminals. So, for a non-terminal A in the definition of $\text{FIRST}(A)$ used by `gtb` the set $\text{FIRST}_T(A)$ is replaced by the set

$$\text{FIRST}_{\mathbf{T}\cup\mathbf{N}}(A) = \begin{cases} \{x \in \mathbf{T}\cup\mathbf{N} \mid A\overset{+}{\Rightarrow}x\beta,\ \text{for some } \beta\} \cup \{\epsilon\} & \text{if } \alpha\overset{*}{\Rightarrow}\epsilon \\ \{x \in \mathbf{T}\cup\mathbf{N} \mid A\overset{+}{\Rightarrow}x\beta,\ \text{for some } \beta\} & \text{otherwise.} \end{cases}$$

The effect of requiring $A\overset{+}{\Rightarrow}x\beta$ is that A belongs to its own FIRST set if and only if A is left recursive, and thus `gtb` can detect and report on left recursion in a grammar as a side effect of the FIRST set construction.

For example, for the above grammar, ex1, the `gtb` FIRST set for S is

$$\{a, b, S, E\}$$

Similarly, for a grammar symbol x in the definition of $\text{FOLLOW}(x)$ used by `gtb` the set $\text{FOLLOW}_T(x)$ is replaced by the set

$$\text{FOLLOW}_{\mathbf{T}\cup\mathbf{N}}(x) = \begin{cases} \{y \in \mathbf{T}\cup\mathbf{N} \mid S\overset{*}{\Rightarrow}\alpha xy\beta,\ \text{for some } \alpha,\beta\} \cup \{\$\} & \text{if } S\overset{*}{\Rightarrow}\alpha x \\ \{y \in \mathbf{T}\cup\mathbf{N} \mid S\overset{*}{\Rightarrow}\alpha xy\beta,\ \text{for some } \alpha,\beta\} & \text{otherwise.} \end{cases}$$

1.4.5 Examining a `gtb` grammar

Using the `write` method, we can get `gtb` to print out some of the data structures that it has constructed. If we add the line

```
write[ex1_grammar]
```

to the `gtb` script it will print out information about the grammar `ex1_grammar` that it has constructed. (To get more information we switch into verbose mode). By default the output is printed to the screen.

For example, if we input the script

```
(* ex1 *)

S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .

(* the LC commands are enclosed in parantheses *)
```

```
(
ex1_grammar := grammar[S]
gtb_verbose := true
write[ex1_grammar]
)
```

diagnostic information is printed on the screen. We now give these diagnostics, interspersed with some explanation.

```
Grammar report for start rule S
Grammar alphabet
   0 !Illegal
   1 #
   2 $
   3 '*'
   4 '+'
   5 'a'
   6 'b'
-------------
   7 E
   8 S

Grammar rules
E ::= 'a' |
       'b' .
S ::= S[0] '+' S[1] |
       S[2] '*' S[3] |
       E[4] .
```

Internally all of the grammar symbols (and other things as we shall see later) are given unique integer numbers which are used both internally and in some of the output structures. This numbering is listed at the top of the output.

Then the grammar rules are listed, and each instance of each non-terminal on the right hand side of a rule is given an instance number.

```
Grammar sets

terminals = {'*', '+', 'a', 'b'}

nonterminals = {E, S}

reachable = {'*', '+', 'a', 'b', E, S}

reductions = {E ::= 'b' . , E ::= 'a' . , S ::= E . ,
S ::= S '*' S . , S ::= S '+' S . }

nullable_reductions = {E ::= 'b' . , E ::= 'a' . , S ::= E . ,
S ::= S '*' S . , S ::= S '+' S . }

start rule reductions = {S ::= E . , S ::= S '*' S . , S ::= S '+' S . }

start rule nullable_reductions = {S ::= E . , S ::= S '*' S . ,
S ::= S '+' S . }
```

To help identify bugs, gtb performs a 'reachability' analysis to determine which of the symbols in the rules can appear in sentential forms of the given start symbol. Then lists of certain 'items' which are output. The roles of these will be discussed later.

```
first('*') = {'*'}
follow('*') = {S}

first('+') = {'+'}
follow('+') = {S}

first('a') = {'a'}
follow('a') = {$, '*', '+'}

first('b') = {'b'}
follow('b') = {$, '*', '+'}

first(E) = {'a', 'b'}
follow(E) = {$, '*', '+'}

first(S) = {'a', 'b', E, S}
follow(S) = {$, '*', '+'}
rhs_follow(S, 0) = {'+'}
rhs_follow(S, 1) = {#}
rhs_follow(S, 2) = {'*'}
rhs_follow(S, 3) = {#}
rhs_follow(S, 4) = {#}

End of grammar report for start rule S
```

Finally gtb outputs its versions of the FIRST and FOLLOW sets for each symbol.

We can see that also output, for each non-terminal, are sets called rhs-follow sets. These sets are constructed for each instance of each non-terminal in each grammar rule, and they are the FIRST set of the string which follows the particular instance. So for a rule $B ::= \alpha A[m]\beta$ we have

$$\texttt{rhs\_follow}(B, m) = \textsc{first}(\beta)$$

Note: the FIRST sets and rhs-follow sets depend only on the grammar rules, but the FOLLOW sets depend on the start symbol of the grammar. For example, we can construct a different grammar from the rules in our example above by taking E to be the start symbol.

If we input the script

```
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .

(
ex1_grammar := grammar[E]
write[ex1_grammar]
generate[ex1_grammar 10 left sentences]
generate[ex1_grammar 15 right sentential_forms]
)
```

the following output is printed on the screen.

```
Grammar report for start rule E
Grammar alphabet
   0 !Illegal
   1 #
   2 $
   3 '*'
   4 '+'
   5 'a'
   6 'b'
-------------
   7 E
   8 S

Grammar rules
E ::= 'a' |
       'b' .
S ::= S[0] '+' S[1] |
       S[2] '*' S[3] |
       E[4] .

Grammar sets

terminals = {'*', '+', 'a', 'b'}

nonterminals = {E, S}

reachable = {'a', 'b', E}

reductions = {E ::= 'b' . , E ::= 'a' . , S ::= E . ,
S ::= S '*' S . , S ::= S '+' S . }

nullable_reductions = {E ::= 'b' . , E ::= 'a' . , S ::= E . ,
S ::= S '*' S . , S ::= S '+' S . }

start rule reductions = {E ::= 'b' . , E ::= 'a' . }

start rule nullable_reductions = {E ::= 'b' . , E ::= 'a' . }

first('*') = {'*'}
follow('*') = {}

first('+') = {'+'}
follow('+') = {}

first('a') = {'a'}
follow('a') = {$}
```

```
first('b') = {'b'}
follow('b') = {$}

first(E) = {'a', 'b'}
follow(E) = {$}

first(S) = {'a', 'b', E, S}
follow(S) = {}
rhs_follow(S, 0) = {'+'}
rhs_follow(S, 1) = {#}
rhs_follow(S, 2) = {'*'}
rhs_follow(S, 3) = {#}
rhs_follow(S, 4) = {#}

End of grammar report for start rule E

Generated sentences using leftmost derivation

     1: a
     2: b

Generated sentential forms using rightmost derivation

     1: E
     2: 'a'
     3: 'b'
```

1.5 Enumeration and the rules tree

`gtb` represents the grammar, `grammar[S]`, that it constructs using a rules tree. The internal rules tree can be output to a file in VCG [San95] format, allowing it to be viewed. To do this we open a file named, for example `rules.vcg`, using the method

`rules_file := open["rules.vcg"]`,

and then use the method

`render[rules_file my_grammar]`

to output the VCG format to the file `rules.vcg`. The graph can then be viewed by running VCG. Usually this is done by typing a command line instruction such as `vcg rules.vcg`.

The script

```
(* ex1 *)
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .

(
ex1_grammar := grammar[S]

rules_file := open["rules.vcg"]
render[rules_file ex1_grammar]
)
```

creates a rules tree for ex1 that is displayed in VCG as

The nodes in the rules tree are labelled with what are sometimes called LR(0) items, and which in `gtb` are referred to as *slots*. There is a slot for the start of each rule and a slot for each position before and after each symbol in each alternate of the rule.

Each slot has a number which belongs to the enumeration mentioned above. So, for example ex1 above, the integers 3 to 8 are used for the grammar symbols (as listed at the top of the output in the previous section) and the slot numbers start at 9. In addition to its use for numbering the nodes in the rules tree, this number is used frequently in later structures as part of the numbering of other graph nodes and parse actions.

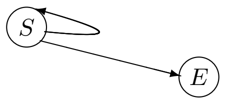
1.6 Grammar dependency graphs

Some parsing techniques view a grammar as a system for string matching in which there are notional pointers into both the grammar and the input string. Initially the grammar pointer points to the left hand side of the start rule and the string pointer points to the left hand end of the input string. At any stage in the parse if the grammar pointer is pointing at a terminal symbol then that symbol is matched to the symbol pointed to by the string pointer, and if the symbols match then both pointers are moved on one place. If the grammar pointer is pointing to a non-terminal on the right hand side of rule then the pointer is moved to the left hand side of the rule for that symbol.

More sophisticated machinery is required to decide where to move the grammar pointer when it is at the left hand side of a rule, and this leads to different forms of parsing technique. But we can see that there is a sense in which a non-terminal on the left hand side of a rule calls non-terminals which appear on the right hand side of the rule.

The relationship A *depends on* B which is defined by the property that B appears on the right hand side of the rule for A turns out to be useful in certain situations. For example, the first step in building a reduction incorporated parser (see Chapter 4) is to construct a new grammar which does not contain any proper self embedding. (A grammar contains self embedding if there is a terminal A and a derivation $A \overset{+}{\Rightarrow} \sigma A \tau$, see below.)

The relation A depends on B is represented in a *grammar dependency graph* (GDG). For example, the GDG for the example grammar, ex1, above is



1.6.1 GDGs in `gtb`

We can get `gtb` to construct the GDG for a grammar using the method

```
my_gdg := gdg[my_grammar]
```

which takes the grammar named `my_grammar` and creates a graph named `my_gdg`. The GDG produced by `gtb` contains more information than just the basic dependency relationships. The edge from A to B is labelled l or r if there is a rule $A ::= B\beta$ or $A ::= \alpha B$, respectively, and the edge is also labelled L or R if there is a rule $A ::= \alpha B \beta$ where $\alpha \neq \epsilon$ or $\beta \neq \epsilon$, respectively.

Formally, an edge of the GDG, from A to B say, is labelled with a subset, $Ł_{A,B}$, of the set $\{L, R, l, r\}$, as follows.

- ◇ If there is a rule $A ::= \alpha B \beta$ where $\alpha \neq \epsilon$ then $L_{A,B}$ contains L.
- ◇ If there is a rule $A ::= \alpha B \beta$ where $\beta \neq \epsilon$ then $L_{A,B}$ contains R.
- ◇ If there is a rule $A ::= \alpha B \beta$ where $\alpha \overset{*}{\Rightarrow} \epsilon$ then $L_{A,B}$ contains l.
- ◇ If there is a rule $A ::= \alpha B \beta$ where $\beta \overset{*}{\Rightarrow} \epsilon$ then $L_{A,B}$ contains r.

As for the rules tree, the internal GDG can be output in VCG format. To do this we open a file named, for example `gdg.vcg`, using the method

```
gdg_file := open["gdg.vcg"],
```

and then use the method

```
render[gdg_file my_gdg]
```

to output the VCG format.

```
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .

(
ex1_grammar := grammar[S]
write[ex1_grammar]

gdg_file := open["gdg.vcg"]
ex1_gdg := gdg[ex1_grammar]
render[gdg_file ex1_gdg]
close[gdg_file]
)
```

Running the command `vcg gdg.vcg` displays the following graph.

It is possible to nest the method calls in the obvious way, so we can also write

```
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .

(
ex1_grammar := grammar[S]
render[open["gdg.vcg"] gdg[ex1_grammar]]
)
```

1.6.2 Recursive non-terminals

A non-terminal A is said to be *recursive* if there is a derivation $A \overset{+}{\Rightarrow} \sigma A \tau$, i.e. if a string that contains A can be derived from A in a derivation that contains at least one step.

If A is recursive then we have a derivation

$$A \Rightarrow \sigma_1 A_1 \tau_1 \Rightarrow \sigma_2 A_2 \tau_2 \Rightarrow \ldots \Rightarrow \sigma_{m-1} A_{m-1} \tau_{m-1} \Rightarrow \sigma A \tau$$

and thus there is a corresponding cycle

$$A \rightarrow A_1 \rightarrow A_2 \rightarrow \ldots A_{m-1} \rightarrow A$$

in the GDG. Conversely if there is a non-empty cycle from A to itself in the GDG then A is recursive. Thus we can identify recursive non-terminals by identifying cycles in the GDG.

For example the GDG for the grammar ex0

$$\begin{array}{lcl} S & ::= & B\ A\ a \mid B\ B \\ A & ::= & B\ b\ A\ B \mid a \\ B & ::= & S\ a\ a \mid \epsilon \mid D \\ D & ::= & d \end{array}$$

has GDG

We see that there are cycles from S, A and B to themselves, correctly reflecting the fact that these non-terminals are all recursive.

A non-terminal A is said to be *left recursive* if there is a derivation $A\overset{+}{\Rightarrow}A\tau$ and *right recursive* if there is a derivation $A\overset{+}{\Rightarrow}\sigma A$.

It is not hard to see that A is left recursive if and only if there is a derivation of the form

$$A\Rightarrow\sigma_1A_1\tau_1\overset{*}{\Rightarrow}A_1\tau_1 \Rightarrow\sigma_2A_2\tau_2\overset{*}{\Rightarrow}\ldots\overset{*}{\Rightarrow}\sigma_{m-1}A_{m-1}\tau_{m-1}\overset{*}{\Rightarrow}A_{m-1}\tau_{m-1}\Rightarrow\sigma A\tau\overset{*}{\Rightarrow}A\tau$$

where $\sigma_1,\sigma_2,\ldots,\sigma_{m-1},\sigma\overset{*}{\Rightarrow}\epsilon$. In particular there are rules

$$A\Rightarrow\sigma_1A_1\tau_1, \qquad A_1\Rightarrow\sigma_2A_2\tau_2, \quad \ldots, \quad A_{m-1}\Rightarrow\sigma A\tau.$$

so the edges $A\to A_1$, $A_1\to A_2$, …, $A_{m-1}\to A$ will all have the label l on them. Thus we have that A is left recursive if and only if there is a non-empty cycle from A to itself in the GDG with the property that every edge in the cycle has a label that includes l.

In the above example there are paths, all of whose edges are labelled l, from S and B to themselves, correctly reflecting the fact that S and B are left recursive but A is not.

Similarly a non-terminal A is right recursive if and only if there is a non-empty cycle from A to itself in the GDG with the property that every edge in the cycle has a label that includes r. In the above example the edge from B to S is not labelled r and we see that S and B are not right recursive, but A is right recursive.

A non-terminal A is said to be *self embedding* if there is a derivation $A\overset{+}{\Rightarrow}\sigma A\tau$ where $\sigma, \tau \neq \epsilon$. It is not hard to see that a non-terminal A is self embedding if and only if there is a path in the GDG from A to itself in which at least one edge has the label L and at least one edge has the label R.

We can see from the above GDG that all the non-terminals in the above grammar ex0 are self embedding.

Chapter 2

LR automata

As we have already mentioned, it is common to take a grammar Γ and a string u and to try to determine whether or not $u \in L(\Gamma)$. A well known technique which is often used for this is called *shift reduce parsing*. In its standard form a shift reduce parser is a bottom up parser which, if it succeeds, produces a right-most derivation of the input string u.

The idea is to have an algorithm which, for a given grammar, takes as input a string $X_1X_2\dots X_n$, say, of terminals *and* non-terminals and which produces as output another string $Y_1Y_2\dots Y_m$, if one exists, such that

$$S \overset{*}{\Rightarrow} Y_1Y_2\dots Y_m \Rightarrow X_1X_2\dots X_n.$$

Such an algorithm is applied to an input string, then successively to its own output with the aim of obtaining the string containing just the start symbol, S.

Of course, the problem of determining whether the string $X_1X_2\dots X_n$ can be derived from the start symbol is the point of the whole process, so we can't expect to be able to tell whether a string $Y_1\dots Y_m$ with the above property exists. If we could then we wouldn't need the process at all.

It turns out that it is possible to construct a finite state automaton with the property that it accepts an input string if and only if it is an initial segment of a sentential form in the grammar and, as we shall see, this is enough. These automata are the standard LR automata which form the basis of the stack based parsing technique introduced by Knuth [Knu65].

In this chapter we shall describe a non-standard construction of the LR automata, constructing initial NFAs and then applying the subset construction to generate the traditional DFAs. This is based on the approach described by Grune [GJ90], and we believe that it gives important pedagogic insight into the structure and behaviour of these automata. For this reason `gtb` constructs the LR automata in precisely this stepwise fashion (rather than in the more direct *move* and *closure* based approach described in standard texts such as [ASU86]).

2.1 State machines for finding derivations

In this section we describe how to construct an NFA which works as follows:

◇ It reads part of the input string $X_1 \dots X_i$, say, and then stops.

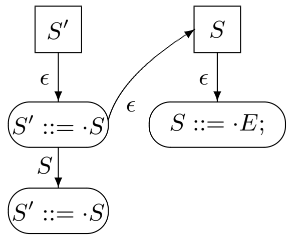
◇ When it stops it either reports an error, or there is some string α such that $S \overset{*}{\Rightarrow} X_1 \dots X_i \alpha$ and there is a production $Z ::= X_j X_{j+1} \dots X_i$. In the latter case, the string $X_1 \dots X_{j-1} Z X_{i+1} \dots X_n$ is returned. Replacing the right hand side of a rule with its left hand side in this way is called a *reduction*.

We shall illustrate the construction and operation of the NFA using the following grammar ex2 which has a special augmented start rule $S' ::= S$.

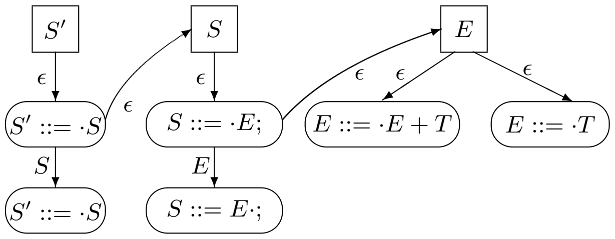
$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & E; \\ E & ::= & E + T \mid T \\ T & ::= & 0 \mid 1 \end{array}$$

The start node of our NFA is a node labelled with S' the augmented start symbol. We are looking to construct a string just containing S and we describe this state using the notation $\cdot S$. We create a node labelled $S' ::= \cdot S$ and an arrow labelled ϵ to it from the start node. If the input string is just S then we have a complete, single step derivation and we can terminate and accept the string. We achieve this by creating an accepting node, labelled $S' ::= S\cdot$ to indicate that we have seen the string S, and an arrow labelled S to this node.

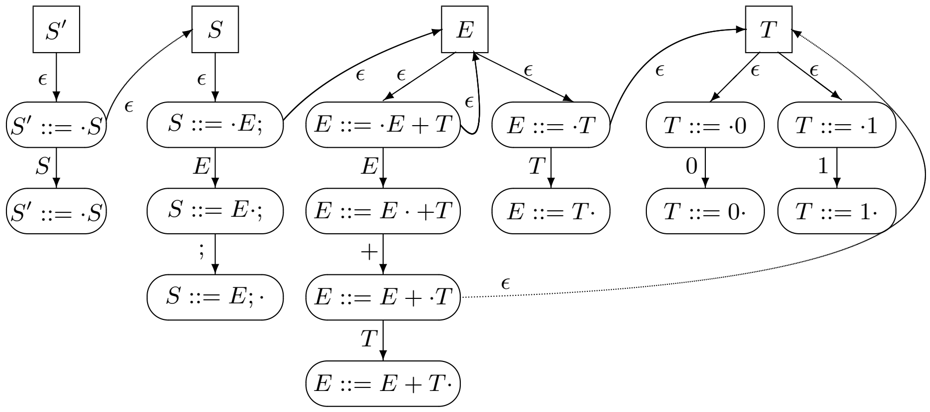
If the input string is not S then we are looking to construct it. To construct S we need to find one of the alternates on the right-hand-side of a production rule for S. We create a new special header node labelled S which has a child node for each alternate of the rule for S (in this case E;). We put a dot in front of each of the alternates to indicate that we are now looking for that string. Since the move from S to one of it's alternates doesn't consume any of the input string the arrow between these nodes is labelled ϵ.



Now we are looking for E. If we read the next symbol of the input and it is E then we can move on and look for the next symbol, in this case a semicolon. Otherwise we need to look to construct an E. We build this into the NFA by making a header node labelled E. Then, as for S, we need to look for some alternate in the production rule for E, so we add new nodes and epsilon-transitions as for S.



We carry on this construction process until all the branches of the NFA terminate in accepting states. The complete NFA for the above grammar is



2.1.1 Formal NFA construction

Formally we construct the LR(0) NFA from a grammar Γ as follows.

1. For each non-terminal A in the augmented grammar construct a header node labelled A.

2. For each rule $A ::= x_1 \dots x_d$

   (a) create nodes labelled $A ::= x_1 \dots x_{i-1} \cdot x_i \dots x_d$, for $1 \leq i \leq d+1$,

   (b) create an edge labelled ϵ from the node labelled A to the node labelled $A ::= \cdot x_1 \dots x_d$,

   (c) create an edge labelled x_i from the node labelled $A ::= x_1 \dots x_{i-1} \cdot x_i \dots x_d$ to the node labelled $A ::= x_1 \dots x_i \cdot x_{i+1} \dots x_d$, for $1 \leq i \leq d$.

   (d) if x_i is a non-terminal create an edge labelled ϵ from the node labelled $A ::= x_1 \dots x_{i-1} \cdot x_i \dots x_d$ to the header node labelled x_i, for $1 \leq i \leq d$.

3. The start state is the state labelled with the augmented start symbol.

4. The accepting states are the states with a label of the form $A ::= \alpha \cdot$.

2.1.2 Using an LR(0) NFA to parse

Standard shift reduce parsers use push down automata, i.e. a stack is used together with the FA. Ultimately this is how our shift reduce parsers will work but it is instructive to consider parsing directly with just the LR(0) NFA because this highlights the role of the stack and informs our development of reduction incorporated parsers in Chapter 4.

Formally we have an LR(0) NFA together with an action which is to be carried out when the NFA reaches an accepting state. This action will be to replace a substring of the input string by a non-terminal.

Given an input string $X_1X_2 \ldots X_m$ we begin in the start state of the NFA. Then at each stage we either move along an ϵ arrow into another state or we read the next input symbol and move into a new state along an arrow labelled with that symbol, if one exists. If no moves are possible and the current NFA state is not a leaf state then the parse has failed. If the NFA moves into a leaf state (one with no out-transitions) the string on the right hand side of the item labelling this state will occur in the input string. Replace this string with the non-terminal on the left hand side of the item, and return the result.

Suppose that we are attempting to parse the string $0 + 1;$ in the above grammar, and that we have so far constructed $E + 1;$. Thus we have the derivation steps $E + 1; \Rightarrow T + 1; \Rightarrow 0 + 1;$. To construct the previous step, starting in state 0 we run the NFA on $E + 1;$.

```
state      input              action
  0      ^E + 1 ;        move to state 1
  1      ^E + 1 ;        move to state 3
  3      ^E + 1 ;        move to state 4
  4      ^E + 1 ;        move to state 5
  5      ^E + 1 ;        move to state 6
  6      ^E + 1 ;        read input symbol
  7       E^+ 1 ;        read input symbol
  8       E +^1 ;        move to state 12
  12      E +^1 ;        move to state 15
  15      E +^1 ;        read input symbol
  16      E + 1^;        replace 1 by T, return string E + T ;
```

We now have the steps $E+T; \Rightarrow E+1; \Rightarrow T+1; \Rightarrow 0+1;$ and we can run the NFA again on the input $E + T;$.

2.1.3 Grammar augmentation in `gtb`

If it is necessary for a grammar to be augmented, then a `gtb` function that requires an augmented grammar will test its input and augment it if necessary. However, augmented versions of grammars are different from the corresponding original grammar. FIRST and FOLLOW sets, and perhaps more importantly the internal enumeration of the symbols and slots, are different for the augmented version of a grammar.

It is possible to explicitly instruct `gtb` to augment a grammar using the method

```
augment_grammar[my_grammar]
```

It is important to note that this modifies the grammar `my_grammar`, it does not create a new, independent grammar. It is also important to note that a grammar is only augmented if its start rule is not already of the form $S ::= A$, where S is not recursive. (In other words, if the grammar is already augmented.)

The following script, `ex2.gtb`, creates the grammar in the above example,

```
(* ex2 Augmentation *)
S ::=  E ';' .
E ::= E '+' T | T .
T ::= '0' | '1' .

(
ex2_grammar := grammar[S]
render[open["rules1.vcg"] ex2_grammar]

augment_grammar[ex2_grammar]
render[open["rules2.vcg"] ex2_grammar]
)
```

The first time that the `render` function is called it generates the rules tree for the original unaugmented grammar. The VCG representation of this rules tree is

The second time it is called, the grammar has been augmented, so the following is now the rules tree.

2.1.4 NFAs in `gtb`

We can get `gtb` to build an LR(0) NFA from a grammar, `my_grammar` say, that we have already built from the input rules. The basic form of the method call is

```
my_nfa := nfa[my_grammar lr 0]
```

There are various forms of NFA which can be constructed and thus the `nfa` method is parameterised by both the input grammar and various other options. Many of the options have defaults but it is necessary to specify the type of NFA required, in this case lr, and then the level of lookahead, in this case 0. This constructs an NFA of the type we have described above. If we render this NFA to a file `gtb` generates VCG output which can be displayed graphically.

From the following script

```
(* ex2 *)
S ::=  E ';' .
E ::= E '+' T | T .
T ::= '0' | '1' .

(
ex2_grammar := grammar[S]

ex2_nfa := nfa[ex2_grammar lr 0]
render[open["nfa.vcg"] ex2_nfa]
)
```

generates a VCG file, `nfa.vcg`, which contains the NFA. The function `nfa[ex2_grammar lr 0]` automatically augments the grammar as described in Section 2.1.3.

The NFA generated by the above example is a little large to reproduce in these notes, so we show the effect using a smaller grammar.

Consider the grammar, ex3,

$$S \ ::= \ A\ b \mid a\ d \qquad\qquad A \ ::= \ A\ a \mid \epsilon$$

The script

```
(* ex3  Generating NFAs *)
S ::=  A 'b' | 'a' 'd' .
A ::=  A 'a' | # .

(
ex3_grammar := grammar[S]
ex3_nfa := nfa[ex3_grammar lr 0]
render[open["nfa.vcg"] ex3_nfa]
render[open["rules.vcg"] ex3_grammar]
)
```

generates the following VCG NFA graph.

Again the grammar is automatically augmented by the `nfa` function, so the corresponding rules tree is now

The node numbering in the NFA is based on the slot numbering in the rules tree. The NFA header nodes have numberings in the running enumeration maintained by `gtb` (but the other NFA nodes do not). So, in the above example, the NFA header node labelled S is the 25th element in the enumeration, and the slot number in the rules tree for S is 11. So the corresponding NFA node is labelled 11.25. The descendents of this node are also labelled in the form $25.m$, where m is the corresponding slot number in the rules tree.

2.2 DFAs and stacks

There are two obvious inefficiencies with the NFA based parser described in the previous section: the NFA is non-deterministic and thus if a particular parse is unsuccessful it may be necessary to back-track and try a different sequence of steps, and every time a non-terminal replacement is carried out the whole input string is read again.

We can improve the first case by using the subset construction (given, for example, in [ASU86]) to construct a deterministic finite state automaton that is equivalent to the LR(0) NFA, and we can solve the second problem by using a stack.

The following terminology is used for certain structures that are used in the subset construction. We give the terminology here because it is used in the diagnostic reporting carried out by `gtb`.

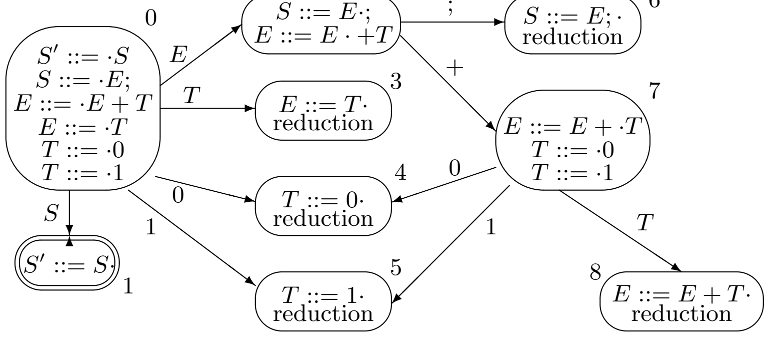
A production $A ::= \gamma \cdot \alpha$ which has a 'dot' somewhere on its right hand side is called an *LR(0)-item*, (in `gtb` it is also referred to as a slot as described above).

If P is a set of items the *ϵ-closure* of P, $cl(P)$, is the smallest set which contains P and all items of the form $B ::= \cdot\beta$ where there is item of the form $A ::= \gamma \cdot B\alpha$ in P.

If P is a set of items and X is a terminal or non-terminal then P_X is the set of all items $A ::= \gamma X \cdot \alpha$ such that $A ::= \gamma \cdot X\alpha$ is in P. We define $move(P, X) = cl(P_X)$.

The DFA constructed from the LR(0) NFA using the subset construction is called the LR(0) DFA. We label the DFA states with the union of the labels of the NFA states from which the DFA state has been constructed.

Applying the subset construction to the NFA from the end of Section 2.1 gives



The accepting states of the LR(0) DFA are those states whose label includes an item of the form $A ::= \alpha\cdot$.

A *grammar is LR(0)* if, in the LR(0) DFA constructed as above, the labels of the accepting states have only one item in them.

We can use a DFA to parse a string in a similar fashion to the use of the NFA. However, to avoid re-reading the input string, each time we move to a DFA state this state is pushed onto a stack. When a state labelled with an item $A ::= x_1 \dots x_n\cdot$ (a *reduction item*) is reached, the top n states are popped of the stack, leaving k say on top, and we move to the state which is the target of the transition from k labelled A.

2.2.1 LR(0) tables

In practice it is usual to write core data structure of an LR parser as a table. The rows of the table are indexed by the DFA state numbers and the columns are indexed by \$, the terminals and the non-terminals. The columns indexed by the terminals form what is usually called the *action* part of the table, and the columns indexed by the non-terminals form the *goto* part of the table.

Formally the LR(0) table is constructed as follows.

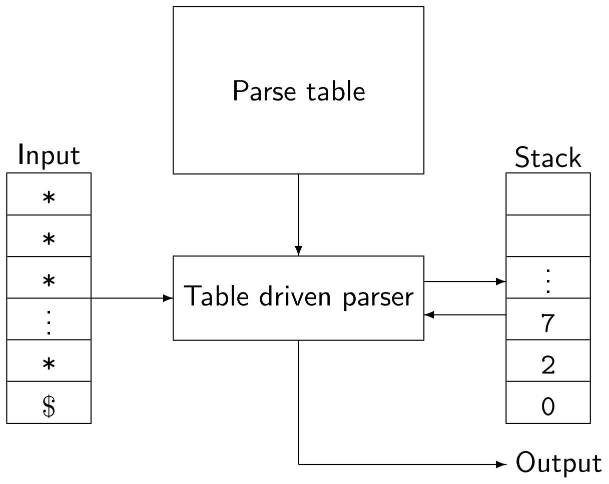
- ◇ If there is a DFA transition labelled with a terminal a from state h to state k then sk is put in row h, column a of the table. These actions are referred to as *shifts*.
- ◇ If there is a DFA transition labelled with a non-terminal A from state h to state k then gk is put in row h, column A of the table.
- ◇ If the label of state h includes the item $A ::= x_1 \dots x_n\cdot$, where $A ::= x_1 \dots x_n$ is rule m and $A \neq S'$, put rm in all the columns labelled with terminals, and the \$ column, of row h.
- ◇ If the label of state h includes the item $S' ::= S\cdot$ put *acc* the \$ column of row h.

The following is the LR(0) table for the grammar

$$\begin{array}{llll@{\qquad}llll}
0. & S' & ::= & S & 3. & E & ::= & T \\
1. & S & ::= & E; & 4. & T & ::= & 0 \\
2. & E & ::= & E+T & 5. & T & ::= & 1
\end{array}$$

	\$	0	1	+	;	S	E	T
0	-	s4	s5	-	-	g1	g2	g3
1	acc	-	-	-	-	-	-	-
2	-	-	-	s7	s6	-	-	-
3	r3	r3	r3	r3	r3	-	-	-
4	r4	r4	r4	r4	r4	-	-	-
5	r5	r5	r5	r5	r5	-	-	-
6	r1	r1	r1	r1	r1	-	-	-
7	-	s4	s5	-	-	-	-	g8
8	r2	r2	r2	r2	r2	-	-	-

2.2.2 Parsing with an LR table



We begin with the start state, 0, on the stack. At each stage in the parse, the parser looks at the state, h say, on the top of the stack and the next input symbol, a. An entry is then selected from row h, column a of the table. If there is no entry the parse stops and an error message is given. If the entry is sk then the parser pushes k onto the stack and reads the next input symbol. If the entry is rm then rule m, $A ::= x_1 \dots x_d$ say, is found, and d symbols are popped off the stack, leaving t say on the top of the stack. The entry, gk say, in row t, column A of the table is then fetched and k is pushed onto the stack. If the action is acc then if the input symbol is \$ the parser terminates and reports success.

Example Parse $0 + 1;$.

```
stack                 remaining input   next action
0                      0 + 1 ; $         s4
0 4                    + 1 ; $           r4
0 3                    + 1 ; $           r3
0 2                    + 1 ; $           s7
0 2 7                  1 ; $             s5
0 2 7 5                ; $               r5
```

```
0 2 7 8                ; $              r2
0 2                    ; $              s6
0 2 6                  $                r1
0 1                    $                acc
return success
```

The string of input symbols which is replaced by a non-terminal when a reduction is carried out (i.e. the string of grammar symbols popped off the stack) is called a *handle*.

2.2.3 DFAs and LR parsers in `gtb`

For pedagogic purposes, `gtb` constructs the LR(0) DFA from the NFA as described above, using the subset construction. The `gtb` method to do this from a pre-constructed NFA, `my_nfa`, is

```
my_dfa := dfa[my_nfa]
```

Also, as for the NFA, the method `render[dfa_file my_dfa]` produces a VCG format file that can be used to visualise the DFA.

For example, the `gtb` script

```
(* ex3 *)
S ::=  A 'b' | 'a' 'd' .
A ::=  A 'a' | # .

(
ex3_grammar := grammar[S]
ex3_nfa := nfa[ex3_grammar lr 0]

ex3_dfa := dfa[ex3_nfa]
render[open["dfa.vcg"] ex3_dfa]
)
```

generates the following VCG output DFA

`gtb` has an LR parser whose structure is of the form described in Section 2.2.2. The parser is run using a specified DFA and a specified input string using the method

```
lr_parse[my_dfa STRING]
```

Here STRING is the sequence of input symbols, enclosed in double quotes. The script

```
(* ex2 *)
S ::=  E ';' .
E ::= E '+' T | T .
T ::= '0' | '1' .

(
ex2_grammar := grammar[S]
ex2_dfa := dfa[nfa[ex2_grammar lr 0]]
render[open["dfa.vcg"] ex2_dfa]

gtb_verbose := true
lr_parse[ex2_dfa "0+1;"]
gtb_verbose := false
)
```

creates the DFA

In its default form the parse function `lr_parse` reports `accept` or `reject`. However, we can get `gtb` to report a trace of the parser using the `gtb_verbose` mode. We set this using the method

`gtb_verbose := true`

To switch the verbose mode off again we set `gtb_verbose` to `false`. Many of the `gtb` methods have a verbose mode. The above script generates the following output, which we discuss below.

```
******: LR parse: '0+1;'
Lexer initialised: lex_whitespace terminal suppresssed,
lex_whitespace_symbol_number 0
Lex: 4 '0'

Stack: [34]
State 34, input symbol 4 '0', action 35 (S35)
Lex: 3 '+'

Stack: [34] (4 '0') [35]
```

```
State 35, input symbol 3 '+', action 13 (R[2] R23 |1|->10)
Goto state 34, goto action 39

Stack: [34] (10 'T') [39]
State 39, input symbol 3 '+', action 14 (R[3] R24 |1|->7)
Goto state 34, goto action 37

Stack: [34] (7 'E') [37]
State 37, input symbol 3 '+', action 40 (S40)
Lex: 5 '1'

Stack: [34] (7 'E') [37] (3 '+') [40]
State 40, input symbol 5 '1', action 36 (S36)
Lex: 6 ';'

Stack: [34] (7 'E') [37] (3 '+') [40] (5 '1') [36]
State 36, input symbol 6 ';', action 12 (R[1] R22 |1|->10)
Goto state 40, goto action 42

Stack: [34] (7 'E') [37] (3 '+') [40] (10 'T') [42]
State 42, input symbol 6 ';', action 15 (R[4] R28 |3|->7)
Goto state 34, goto action 37

Stack: [34] (7 'E') [37]
State 37, input symbol 6 ';', action 41 (S41)
Lex: EOS

Stack: [34] (7 'E') [37] (6 ';') [41]
State 41, input symbol 2 '$', action 16 (R[5] R29 |2|->8)
Goto state 34, goto action 38

Stack: [34] (8 'S') [38]
State 38, input symbol 2 '$', action 11 (R[0] R21 |1|->9 Accepting)
******: LR parse: accept
```

In the above output the current stack and the next action are shown at each step in the parse. The elements of the form `[n]` on the stack are the DFA state numbers, these numbers can be seen on the VCG version of the DFA. In the above example, initially the stack consists just of the start state, `[34]`.

The elements of the form `(m 'x')` on the stack are grammar symbols, x is the actual symbol and m is its number in the `gtb` generated enumeration, and can be obtained by using the `write[my_grammar]` method as described in Section 1.4.4. (This number is also given in `parse.tbl` which we shall describe below.) It is common in descriptions of LR parsers to include the symbol which labelled the transition from one state on the stack to the next, because this can make it easier both for the reader to see how the process is working, and for the user to find the point in the input where the parse fails, it if fails. Thus after the first action in the above example the stack is of the form

```
[34] (4 '0') [35]
```

with state 34 on the bottom, then the terminal symbol 0, and then the state

35.

The actions are given numbers internally. If the action is a shift then the action number is the number of the state to be pushed onto the stack. If the action is a reduction then the details of the reduction are printed out. The reduce actions are numbered internally by `gtb` in the form `R[n]`. To see which reduction corresponds to `R[n]` we use the parse table, which is written by `gtb`.

If we `write` the DFA a textual version of the parse table corresponding to the DFA is output.

```
write[open["parse.tbl"] my_dfa]
```

creates a file called `parse.tbl` that contains a text version of the `gtb` internal DFA table representation, and the reduction numbers are given in this file.

For our example grammar ex2 the file `parse.tbl` is

```
LR symbol table
0 !Illegal
1 #
2 $
3 +
4 0
5 1
6 ;
7 E
8 S
9 S!augmented
10 T

LR state table
34 0:
 S35
 S36
 S37
 S38
 S39
35 4:
 R[2] {2}
 R[2] {3}
 R[2] {4}
 R[2] {5}
 R[2] {6}
36 5:
 R[1] {2}
 R[1] {3}
 R[1] {4}
 R[1] {5}
 R[1] {6}
37 7:
 S40
 S41
38 8:
 R[0] {2}
 R[0] {3}
 R[0] {4}
 R[0] {5}
 R[0] {6}
39 10:
 R[3] {2}
 R[3] {3}
 R[3] {4}
 R[3] {5}
 R[3] {6}
40 3:
 S35
 S36
 S42
41 6:
 R[5] {2}
 R[5] {3}
 R[5] {4}
 R[5] {5}
 R[5] {6}
42 10:
 R[4] {2}
 R[4] {3}
 R[4] {4}
 R[4] {5}
 R[4] {6}

LR reduction table
R[0] R21 |1|->9 S!augmented ::= S . Accepting
R[1] R22 |1|->10 T ::= '1' .
R[2] R23 |1|->10 T ::= '0' .
R[3] R24 |1|->7 E ::= T .
R[4] R28 |3|->7 E ::= E '+' T .
R[5] R29 |2|->8 S ::= E ';' .
```

In the state table section each state is listed with its state number followed by the number of the symbol that labels the DFA in-transitions to the state. Below this are the actions associated with the state. (The reductions have (the numbers of) associated lookahead symbols which are used later for SLR(1) and LR(1) parsers.) In the reduction table section we have the reduction reference, followed by the slot number of the reduction item, the length of the right hand side of the reduction, and (the number of) the symbol on the left hand side of the reduction rule.

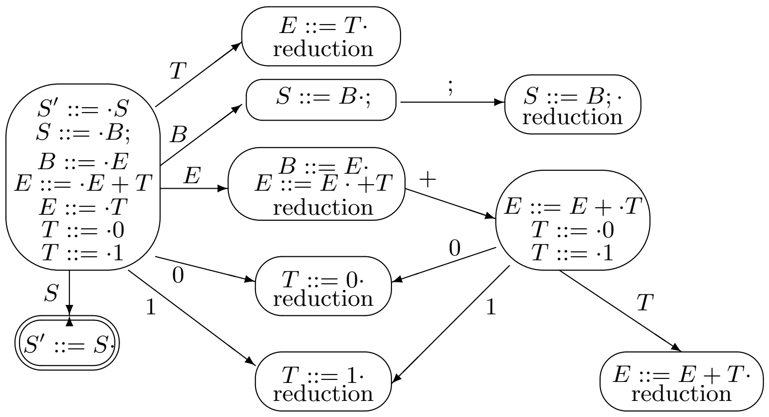
2.3 SLR(1) parse tables

The problem with LR(0) parsing is that a large number of grammars are not LR(0), i.e. their LR(0) parse tables contain some multiple entries. It is possible for an entry to contain a shift action and several reduction actions.

For example, consider the grammar, ex4, which is slight modification of the grammar ex2.

1.	S	$::=$	$B;$	4.	E	$::=$	T
2.	B	$::=$	E	5.	T	$::=$	0
3.	E	$::=$	$E+T$	6.	T	$::=$	1

When we construct the LR(0) DFA for this grammar we get



State 3 contains both a reduction and has a transition to another state. The effect of this is that when we are in state 3 we don't know whether to reduce, popping E off the stack and pushing B, or to read the next input symbol in the hope that it is $+$.

Of course we can make a choice, `gtb` chooses a shift in preference to a reduction and chooses between reductions by taking the first one found. However, if the wrong choice is made then the parser may incorrectly reject the input string.

We can resolve the problem in the above example if we use the next input symbol to decide whether to 'shift' or 'reduce'. If the next input is '+' then we need to push it onto the stack and hope to construct T next. If the next input is ';' then we need to reduce and get B onto the stack. If the input is any other symbol then the parse cannot continue and we report an error.

Using the next input symbol to decide whether to perform a reduction is known as *one symbol lookahead*. Using one symbol lookahead vastly increases the class of grammars which can be correctly parsed using the table based techniques we have been considering.

A simple way in which the input symbol is used to limit reduction application is to use the FOLLOW sets described in Section 1.4.4. This is based on the observation that if we replace a substring α of the current string $\beta\alpha u$ with a non-terminal A then for the result to ultimately be successfully parsed $\beta A u$ must be a sentential form. The next input symbol is the first symbol in u, so, by definition, this symbol must be in FOLLOW(A). Thus we only apply a reduction $A ::= \alpha$ if the next input symbol lies in FOLLOW(A).

Since the FOLLOW sets can be statically computed, we can include this information in the parse table. The resulting table is called an SLR(1) table.

We construct the states exactly as for the LR(0) parse table, and we put *sm*, *gm* and *acc* into the SLR(1) table exactly as for the LR(0) table. For a reduction, if the label of state h includes the item $A ::= x_1 \dots x_n \cdot$, where $A ::= x_1 \dots x_n$ is rule m and $A \neq S'$, we put rm in all the columns of row h that are labelled with elements of FOLLOW(A).

The following in this SLR(1) table for the grammar ex4.

	\$	0	1	+	;	S	B	E	T
0	-	s5	s6	-	-	g1	g2	g3	g4
1	acc	-	-	-	-	-	-	-	-
2	-	-	-	-	s7	-	-	-	-
3	-	-	-	s8	r2	-	-	-	-
4	-	-	-	r4	r4	-	-	-	-
5	-	-	-	r5	r5	-	-	-	-
6	-	-	-	r6	r6	-	-	-	-
7	r1	-	-	-	-	-	-	-	-
8	-	s5	s6	-	-	-	-	-	g9
9	-	-	-	r3	r3	-	-	-	-

A grammar is said to be SLR(1) if the entries in its SLR(1) table each contain at most one element.

In `gtb` the information that is required to build the parse table is collected when the NFA is constructed. So, despite the fact that as automata the NFA and the DFA for an SLR(1) parser are the same as those for the corresponding LR(0) parser, `gtb` generates different internal structures in the two cases. Thus the type of table which is ultimately required is specified in the call to the `gtb` NFA builder.

```
(* ex4 An SLR(1) parser *)
S ::=  B ';' .
B ::= E .
E ::= E '+' T | T .
T ::= '0' | '1' .

(
ex4_nfa := nfa[grammar[S] slr 1]
render[open["nfa.vcg"] ex4_nfa]

ex4_dfa := dfa[ex4_nfa]
lr_parse[ex4_dfa "0+1+1;"]
)
```

The lookahead information (the columns of the table in which a reduction on particular rules should appear) is stored as part of the header nodes in the NFA. For SLR(1) to make the graphs more readable the lookahead information is omitted from the VCG rendering. In the VCG rendering of the DFA the reductions are listed with the appropriate lookahead symbols.

Consider again the grammar, ex3, from Section 2.1.4. The script

```
(* ex3 *)
S ::=  A 'b' | 'a' 'd' .
```

```
A ::=  A 'a' | # .

(
ex3_grammar := grammar[S]
ex3_slr := dfa[nfa[ex3_grammar slr 1]]
render[open["dfa1.vcg"] ex3_slr]
)
```

generates the following VCG graph.

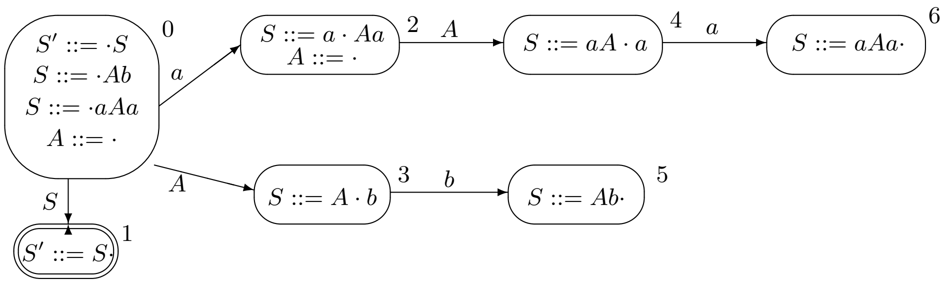
2.4 LR(1) tables

Using the FOLLOW set information to reduce the number of reductions in the table still leaves conflicts for many realistic grammars. It is possible to reduce significantly the number of conflicts by using the lookahead information in a more subtle way.

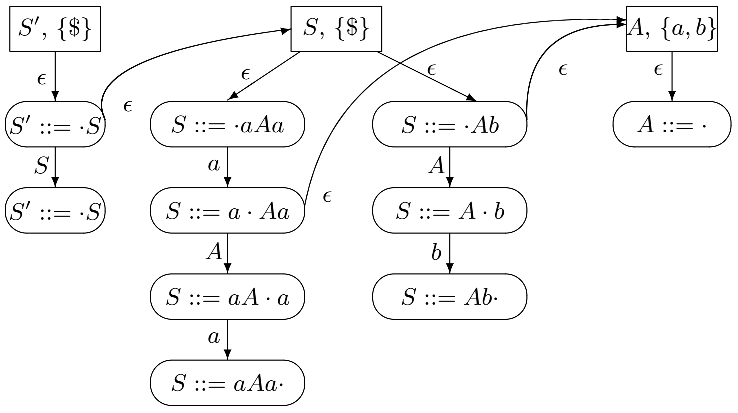
Consider the grammar, ex5,

$$\begin{array}{llllllll} 1. & S & ::= & Ab & \quad 3. & A & ::= & \epsilon \\ 2. & S & ::= & aAa & & & & \end{array}$$

that has LR(0) DFA



The start state of the DFA contains the reduction $A ::= \epsilon$ and a transition labelled a. Since $a \in \text{FOLLOW}(A)$ this results in a conflict in the SLR(1) table. However, if we look at the NFA for ex5



we see that the item $A ::= \cdot$ appears in state 0 as a result of the inclusion of the item $S ::= \cdot Ab$, and thus we really only need to consider performing the reduction $A ::= \epsilon$ if the lookahead symbol is b. This observation motivates the use of 'local' follow set information.

The NFA is constructed on the basis that if we are at a state labelled $A ::= \alpha \cdot B\beta$ then we are ultimately trying to construct A and currently we need B. To construct B we need to match γ, for some γ such that $B ::= \gamma$. When such a rule has been matched then we shall continue to match β, thus we need the next input symbol to be derivable from β. In other words, we wish that when we have matched B that the lookahead symbol is in $\text{FIRST}(\beta)$. Thus instead of creating a header node in the NFA labelled B, we need to create one labelled with B and $\text{FIRST}(\beta)$.

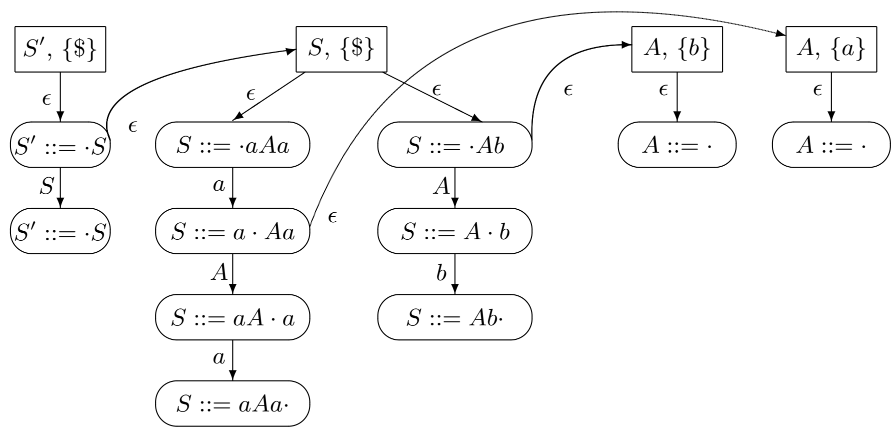
There is a further complication in the case that $\beta \overset{*}{\Rightarrow} \epsilon$. In this case, having matched B we may match A without reading any more input. Thus we need the lookahead symbol to be in the set of lookaheads required by A. Thus, if the header node above the node $A ::= \alpha \cdot B\beta$ is labelled (A, G) then we label the header node for B with $(B, \text{FIRST}(\beta G))$, where $\text{FIRST}(\beta G) = \text{FIRST}(\beta) \cup G$ if $\beta \overset{*}{\Rightarrow} \epsilon$ and $\text{FIRST}(\beta G) = \text{FIRST}(\beta)$, otherwise.

2.4.1 Formal LR(1) NFA construction

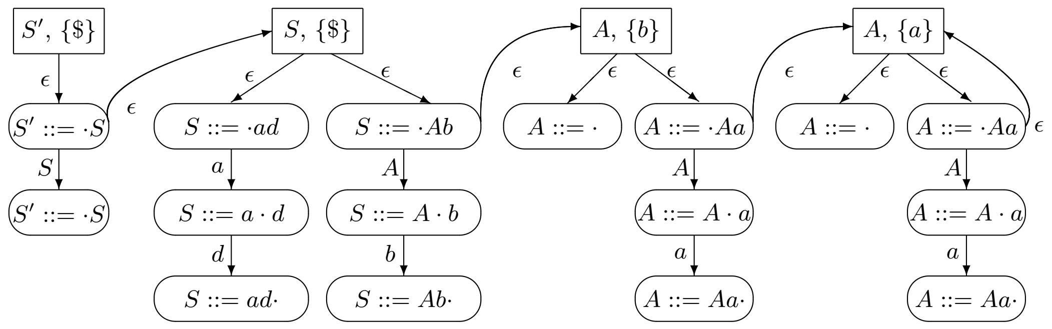
Formally we construct the LR(1) NFA from a grammar Γ as follows.

1. Augment the grammar with a new start symbol, S', create and mark a header node labelled $(S', \{\$\})$. Create a node labelled $S' ::= \cdot S$ and an ϵ-transition to this node from the header node.
2. While there is an unmarked node in the NFA, select an unmarked node h, say, labelled $A ::= \alpha \cdot x\beta$, say, and suppose that the header node above this node is labelled (A, G).
   (a) Create a node, k say, labelled $A ::= \alpha x \cdot \beta$, and a transition labelled x from h to k.
   (b) If $\beta = \epsilon$, mark k.
   (c) If x is a non-terminal, if there is a header node labelled $(x, \textsc{first}(\beta G))$ then add an ϵ-transition from h to this header. Otherwise create a header node, l say, labelled $(x, \textsc{first}(\beta G))$ and an ϵ-transition from h to l. Furthermore, for each grammar rule $x ::= \gamma$ create an NFA node, g say, labelled $x ::= \cdot\gamma$ and an ϵ-transition from l to g. Mark the node l and, if $\gamma = \epsilon$, then mark g.
   (d) mark h.
3. The start state is the header labelled $(S', \{\$\})$.
4. The accepting states are the states with a label of the form $A ::= \alpha\cdot$.

For example, the LR(1) NFA for the grammar ex5 is



and the LR(1) NFA for the grammar ex3 is



We can get `gtb` to build this NFA and the corresponding DFA simply by asking for an LR(1) structure in the method to build the NFA.

```
(* ex3 *)
S ::=  A 'b' | 'a' 'd' .
A ::=  A 'a' | # .

(
ex3_grammar := grammar[S]
render[open["nfa1.vcg"] nfa[ex3_grammar lr 1]]
)
```

2.4.2 LR(1) tables

The LR(1) states are labelled with the so-called LR(1)-items, pairs of the form $(A ::= \alpha \cdot \beta, G)$ where $(A ::= \alpha \cdot \beta)$ is the label of an LR(1) NFA state whose header node is labelled (A, G).

To construct the LR(1) table we put sm, gm and acc into the LR(1) table exactly as for the SLR(1) and LR(0) tables. For a reduction, if the label of state h includes the item $(A ::= x_1 \dots x_n \cdot, G)$, where $A ::= x_1 \dots x_n$ is rule m and $A \neq S'$, put rm in row h, column x for all $x \in G$.

The SLR(1) and LR(1) tables for ex5 are, respectively

SLR(1)	\$	a	b	S	A
0	-	s2/r3	r3	g1	g3
1	acc	-	-	-	-
2	-	r3	r3	-	g4
3	-	-	s5	-	-
4	-	s6	-	-	-
5	r1	-	-	-	-
6	r2	-	-	-	-

LR(1)	\$	a	b	S	A
0	-	s2	r3	g1	g3
1	acc	-	-	-	-
2	-	r3	-	-	g4
3	-	-	s5	-	-
4	-	s6	-	-	-
5	r1	-	-	-	-
6	r2	-	-	-	-

2.4.3 The singleton set model

There are two issues with the DFA construction model described in Section 2.4.1 that we shall now discuss further.

Strictly speaking, applying the subset construction to the LR(1) NFA described above does not always result in the standard LR(1) DFA constructed by the direct method described, for example, in [ASU86]. The DFA is deterministic and correct but it may have more states than the standard DFA.

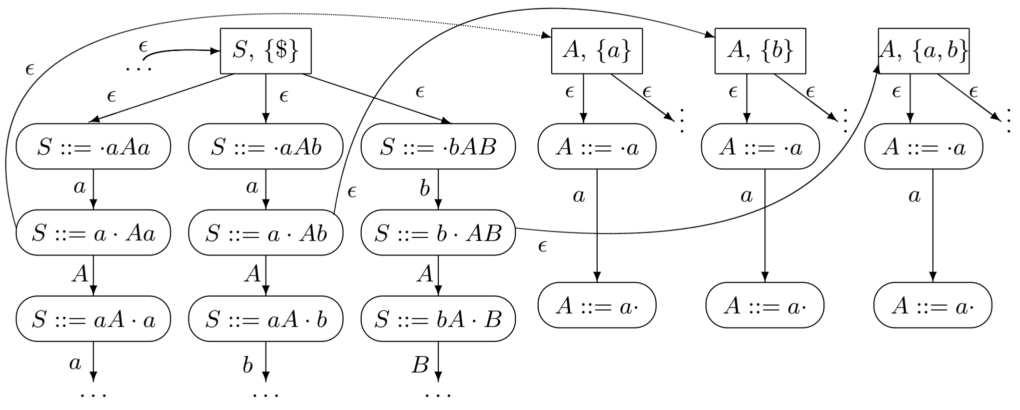
Furthermore, it is possible for each of the possible subsets of a set $\text{FOLLOW}(A)$ to occur as local lookahead sets, thus there may be $(2^{|\text{FOLLOW}(A)|} - 1)$ NFA header nodes for each non-terminal A.

Before discussing the alternative options that can be used in `gtb` to address these problems, we give examples illustrating them.

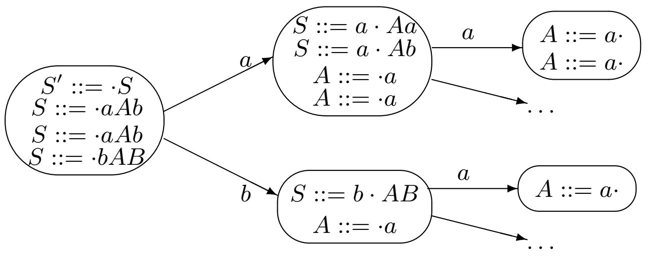
In the first case the issue arises because, under the subset construction, DFA states are made up from sets of NFA states and two DFA states are different if they are not comprised of exactly the same NFA states. When a DFA state is constructed the subset algorithm checks to see if there is already a DFA state with the same set of NFA states and if there is then this DFA state is reused. This ensures that the DFA construction algorithm terminates. Consider the grammar ex6

$$\begin{array}{lll} S & ::= & aAa \mid aAb \mid bAB \\ A & ::= & a \\ B & ::= & a \mid b \end{array}$$

The LR(1) NFA as described above has three headers labelled A. The relevant portion of the NFA is shown below.



If the subset construction is strictly applied to generate the DFA two states containing the reduction $A ::= a$ will be created. The relevant part of the DFA is



In the standard DFA construction these two states would have been merged because they contain the same item with the same lookahead set.

This problem can create many additional DFA states. For example, for the grammar for ISO Pascal included with the distribution of `gtb`, the DFA constructed from the LR(1) NFA using the strict form of the subset construction has 12,258 states while the standard LR(1) DFA has 2,608 states.

`gtb` adopts two approaches to address this problem. The first approach is straightforward: instead of simply applying the subset construction, when an LR(1) NFA is being processed `gtb` accepts two DFA states as being equal if the union of the labels from the corresponding NFA states are the same.

The other approach is to use a different form of NFA. This approach, which we now discuss, also addresses the second issue mentioned above: the potentially large number $(2^{|\text{FOLLOW}(A)|} - 1)$ of header nodes.

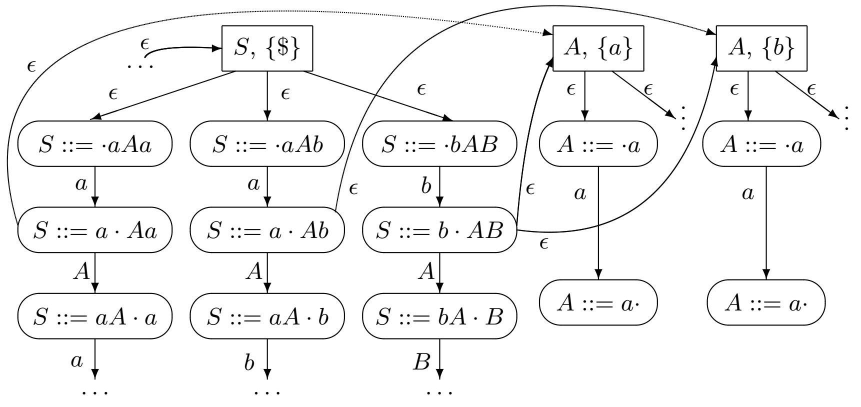
We create a singleton model LR(1) NFA by creating header nodes labelled (A, x) for each non-terminal A and for each $x \in \text{FOLLOW}(A)$. We then build the NFA in a similar fashion to the LR(0) NFA. Given an item $A ::= \alpha \cdot B\beta$ we create a node labelled with this item and ϵ-transitions from this node to the headers (A, x) where $x \in \text{FIRST}(\beta)$.

Singleton follow set LR(1) NFA construction

1. For each non-terminal A in the grammar and for each $x \in \text{FOLLOW}(A)$ construct a header node labelled (A, x).

2. For each rule $A ::= x_1 \dots x_d$ and for each header (A, b)
   (a) Create nodes labelled $A ::= x_1 \dots x_{i-1} \cdot x_i \dots x_d$, for $1 \le i \le d+1$.
   (b) Create an edge labelled ϵ from the node labelled (A, b) to the newly created node labelled $A ::= \cdot x_1 \dots x_d$.
   (c) Create an edge labelled x_i from the node labelled $A ::= x_1 \dots x_{i-1} \cdot x_i \dots x_d$ to the node labelled $A ::= x_1 \dots x_i \cdot x_{i+1} \dots x_d$, for $1 \le i \le d$.
   (d) If x_i is a non-terminal create an edge labelled ϵ from the node labelled $A ::= x_1 \dots x_{i-1} \cdot x_i \dots x_d$ to the header nodes labelled (x_i, y), for each $y \in \text{FIRST}(x_{i+1} \dots x_d b)$, for $1 \le i \le d$.
3. The start state is the state labelled S'.
4. The accepting states are the states with a label of the form $A ::= \alpha \cdot$.

The main fragment of the singleton model LR(1) NFA for ex6 is



Applying the subset construction in its standard form to the singleton model LR(1) NFA results in a DFA that is identical to the LR(1) DFA constructed using the standard construction procedure. Furthermore, the LR(1) NFA has $|\text{FOLLOW}A|$ header nodes for each non-terminal A.

It is not clear whether the singleton model LR(1) NFA is better in practice than the original version, which is referred to as the full subset model in `gtb`. Although in worst case the singleton model has fewer NFA headers, it is likely in practice that the full subset model will have fewer headers. In particular if we consider SLR(1) NFAs, the singleton model has $|\text{FOLLOW}(A)|$ headers while the full subset model has only one header for each non-terminal. Furthermore the singleton model has edges for each element in the local follow sets.

`gtb` allows the user to choose which NFA model they want, allowing the two approaches to be compared. The method

```
my_nfa := nfa[my_grammar lr 1 singleton_lookahead_sets]
```

generates the singleton model LR(1) NFA, while the method

```
my_nfa := nfa[my_grammar lr 1 full_lookahead_sets]
```

generates the original NFA, which is also the default action.

2.5 LALR DFAs

Historically, LR(1) DFAs were considered too large to be practical, in general they have many more states than the SLR(1) DFA for the same grammar.

DeRemer [DeR69, DeR71] described another type of DFA, the LALR DFA, that has the same number of states as the corresponding SLR(1) DFA but, in general, fewer conflicts. The LALR DFA is the one used by YACC and many other standard parser generators.

Conceptually an LALR DFA can be thought of as an LR(1) DFA in which certain states have been merged. If two LR(1) DFA states have labels which differ only in the lookahead sets associated with the items, then these two states are merged, merging the lookahead sets for corresponding items. There is an algorithm for constructing an LALR DFA directly without using the LR(1) DFA but `gtb` uses the conceptually simple merging approach.

The `gtb` method

```
lalr_dfa := la_merge[my_dfa]
```

takes any LR DFA and merges states that differ only in the lookahead sets of the items that label them. To construct an LALR DFA first construct the LR(1) DFA and then run `la_merge`.

```
(* ex3 *)
S ::=  A 'b' | 'a' 'd' .
A ::=  A 'a' | # .

(
ex3_grammar := grammar[S]
ex3_lalr := la_merge[dfa[nfa[ex3_grammar lr 1]]]
render[open["la_dfa.vcg"] ex3_lalr]
)
```

Nowadays programming languages are designed to have grammars that are almost LR(1); their LR(1) tables have relatively few conflicts and those conflicts which do arise are dealt with using semantic checks. However, there is a very large class of grammars whose LR(1) tables contain conflicts, and left to themselves programmers do not naturally design LR(1) grammars.

Thus we consider general parsing techniques which can be used on all grammars. In the next chapter we consider an extension of the LR technique which, when faced with a conflict, pursues all the possibilities in parallel.

Chapter 3

GLR algorithms

The problem with the standard stack based LR parsers described in Chapter 2 is that the LR(1) parse tables for many grammars contain conflicts, and the parser cannot tell which of the possible actions to choose. If the choice made results in the input being rejected then, to be correct, the parser must backtrack to the point where it made the choice and try a different action. This backtracking can result in unacceptable parse times for many grammars.

An alternative approach is, when a choice of action is encountered, to pursue each of the choices in parallel. The naive approach is to make a copy of the current stack(s) for each choice of action and to proceed with each stack in parallel until the parse is complete or no further action exists for that stack. Of course, this does not solve the problem because there are grammars for which this process generates infinitely many stacks.

Tomita [Tom91] devised a method for combining the multiple stacks in such a way that they require at most quadratic space. This allowed him to give a practical, generalised version of the LR parsing algorithm which effectively explores all possible actions in parallel. The resulting multiple stack structure is known as a graph structured stack (GSS), and algorithms which extend the LR parsing algorithm using a Tomita-style GSS are known as GLR parsers.

In reality Tomita's algorithm contains an error which means it is not correct for grammars which contain a certain type of rule. However, this error can be corrected by modifying the input LR parse table.

In this chapter we shall describe the GSS and Tomita's algorithm, and then describe the modification to the LR tables needed to make the algorithm correct. All of the algorithms discussed can be executed in `gtb` and we shall describe the methods required as we proceed.

3.1 Building a GSS

We describe the GSS associated with an LR parse using two examples.

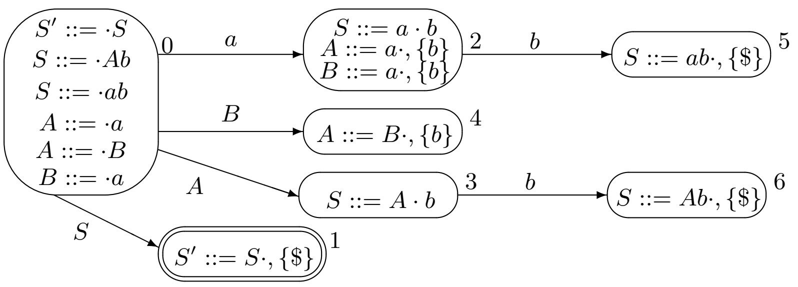
Tomita's original exposition pushes the grammar symbols onto the stack between the state symbols, as mentioned at the end of Section 2.2.3. Part of the role of `gtb` is to implement algorithms as they are written both for pedagogic purposes and to allow the particular features of the algorithms to be studied. Thus the `gtb` implementation of Tomita's algorithm constructs GSSs that contain symbol nodes.

3.1.1 Example grammar ex7

Consider the grammar ex7

$$\begin{array}{lcl} S & ::= & A\ b \mid a\ b \\ A & ::= & a \mid B \\ B & ::= & a \end{array}$$

that has LR(1) DFA



We can use this DFA to recognise the string ab, as follows.

We start in state 0 with 0 on the stack and read the first input symbol, a. We perform the shift action to state 2, giving the stack

```
2
a
0
```

In state 2 we read the next input symbol, b, and find a shift action and two possible reduction actions. The idea is two create two copies of the stack and perform one of the reductions on each stack.

```
2        3        4
a        A        B
0        0        0
```

There is a reduction associated with state 4, but applying this reduction to the corresponding stack $0B4$ generates the stack $0A3$, which already exists. Thus all possible reductions at this stage have been applied.

States 2 and 3 have a shift action on b generating the corresponding stacks

```
5         6
b         b
2         3
a         A
0         0
```

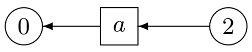
State 4 has no action on input b so the stack with 4 on top dies at this point.

We read the final input symbol, the end-of-string symbol \$. Both states 5 and 6 have reductions, and applying these results the same stack

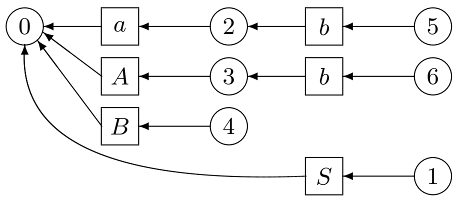
```
1
S
0
```

Since state 1 is the accepting state the string is correctly accepted.

We can represent the stacks as a graph, merging common prefixes. Because of the way the graph is traversed when performing a reduction, we put an edge from each symbol to the symbol *below* it on the stack. For example, we represent the stack $0a2$ as



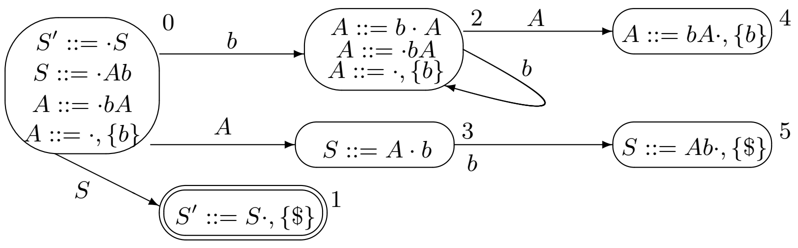
The GSS corresponding to the stack activity in the above example is



3.1.2 Example grammar ex8

$$\begin{array}{lcl} S & ::= & A\ b \\ A & ::= & b\ A \mid \epsilon \end{array}$$

that has LR(1) DFA



We can use this DFA to recognise the string *bbb*, as follows.

We start in state 0 with 0 on the stack. The state 0 has a reduction $A ::= \epsilon$. Since the right hand side is ϵ, nothing is popped off the stack so we simply push A followed by 3 on to the stack. State 3 does not have any reductions so we have two stacks

```
          3
          A
0         0
```

We read the first input symbol, pushing states 2 and 5, respectively, onto the stacks. The lookahead symbol is b so the reduction in state 2 can be applied but the reduction in state 5 is not applied. Finally applying the reduction in state 4 gives the stacks

```
     5    4
     b    A
2    3    2    3
b    A    b    A
0    0    0    0
```

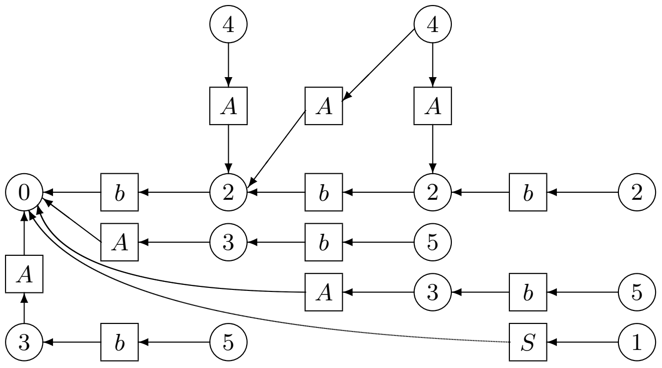
Next we apply the shift action to states 2 and 3. The other two stacks die. Applying the reduction to the stack $0b2b2$ gives the stack $0b2b2A4$, then applying the reduction in state 4 (twice) results in the stacks

```
          4
          A
2    5    2    4
b    b    b    A
2    3    2    2    3
b    A    b    b    A
0    0    0    0    0
```

We apply the last shift, then, since the lookahead symbol is \$, there is one applicable reduction, generating the stacks

```
2
b
2    5
b    b
2    3    1
b    A    S
0    0    0
```

Again we represent these stacks as a graph, merging common prefixes. In this case we have that two of the stacks have the same state, 4, on top at the same step in the process, and such stacks are recombined. (It is this recombination which ensures the the GSS has size which is at most quadratic in the length of the input string.)



The *graph structured stack* associated with an LR parse is the graph obtained by turning each of the possible stacks into a graph with a node for each symbol on the stack an edge from h to k if the symbol corresponding to h is directly above the symbol corresponding to k. These graphs are then merged so that stacks with a common prefix are merged and stacks with the same state above the same element of the input string on the top of the stack are re-combined.

We now discuss Tomita's algorithm that, given a DFA and an input string, constructs the corresponding GSS.

3.2 Tomita's algorithm

In this section we give an informal description of Tomita's algorithm. For detailed discussion and a formal statement of the algorithm see [Tom91], [SJH00] or [SJ06].

The state nodes in the GSS are organised into levels, one level U_i for each input symbol. The nodes in level i correspond to the the tops of stacks that can be obtained by reading the the first i input symbols from the string.

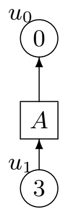
The GSS construction proceeds by performing all possible reduction actions before performing the shift actions and then reading the next input. So the GSS is constructed level by level.

When a GSS state node is constructed the actions associated with the state are collected and stored in a worklist pending application. The shift actions are only performed when there are no pending reduction actions. (The issues surrounding the nature of this worklist are subtle. A full discussion can be found in [SJ06].)

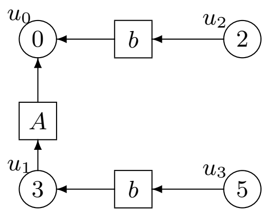
3.2.1 An example

We illustrate Tomita's algorithm using ex8 from Section 3.1.2 and input *bbb*.

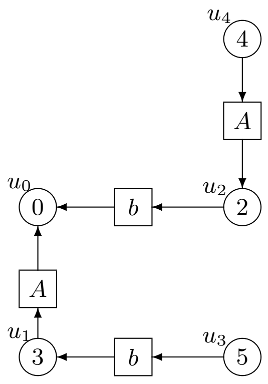
We begin by constructing a GSS node, u_0, labelled 0 and collecting the shift and reduce actions associated with this state. Applying the reduction $A ::= \epsilon$ that has length 0, since the transition labelled A from state 0 goes to state 3, we create a new GSS state node, u_1, labelled 3 and a path from u_1 to u_0 via a new symbol node labelled A.



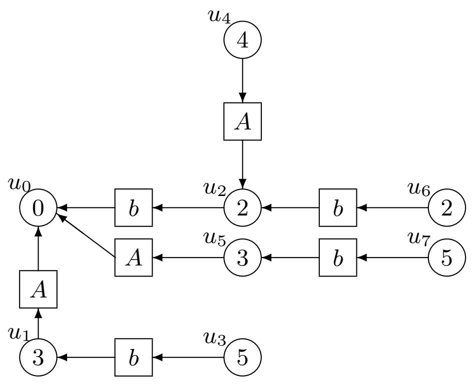
We then collect the shift action associated with state 3. There are no further pending reductions so the construction of U_0 is complete. Next we apply both the pending shift actions, creating new level 1 state nodes, u_2 and u_3, labelled 2 and 5 respectively, and creating paths from these nodes (via symbol nodes labelled a as this is the current input symbol) to u_0 and u_1, respectively.



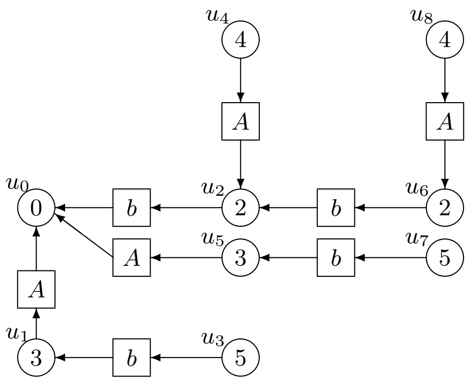
State 2 has a reduction action and applying this we construct a new level 1 node, u_4, labelled 4 and a path from u_4 to u_2 via a new symbol node labelled A.



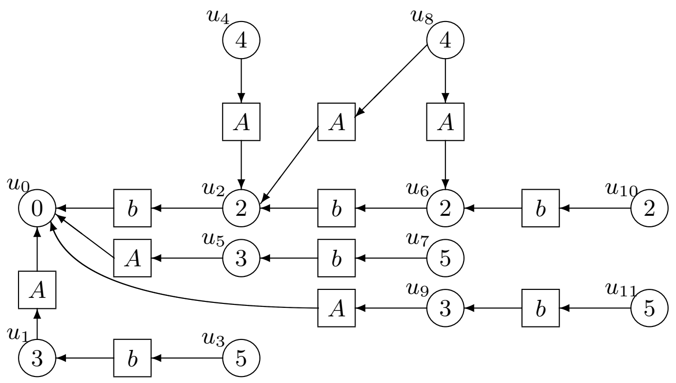
We then apply the reduction associated with state 4. The rule $A ::= bA$ is of length 2 so we trace back along the path of length 4 from u_4, in this case to u_0. Since the label of u_0 is 0 and the transition labelled A from 0 goes to state 3, we create a new level 1 GSS node, u_5, labelled 3 and a path from u_5 to u_0 via a new symbol node labelled A. There are no further pending reductions so we apply the shift actions, resulting in the graph



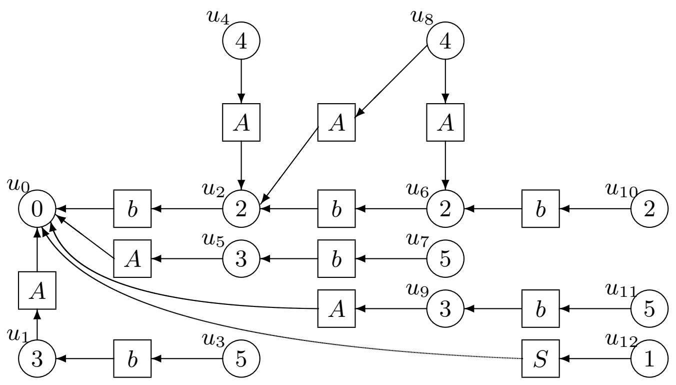
For the reduction $A ::= \epsilon$ in state 2 we create a new level 2 state node, u_8, labelled 4 and a path from u_8 to u_6 via a new state node labelled A.



For the reduction $A ::= bA$ in state 4 we trace back along the path of length 4 from u_8 to u_4 and then, since the transition labelled A from state 2 goes to state 4 and we already have a level 2 node, u_8, labelled 4 we create a path from u_8 to u_4 via a new node labelled A. There is now another path of length 4 from u_8, which goes to u_0. The reduction $A ::= Ab$ must be applied down this path, and we create a new level two node, u_9, labelled 3 and a path from u_9 to u_0 via a new symbol node labelled A. Then we apply the pending shifts.



Because the lookahead symbol is now \$ only state 5 has a reduction. Tracing back along the path of length 4 from u_{11} to u_0, since the transition labelled S from 0 goes to 1 we create a new level 3 node, u_{12}, labelled 1 and a path from u_{12} to u_0 via a new symbol node labelled S. This completes the GSS construction.



Because the last GSS level, level 3, contains a state node, u_{12}, whose label is 1, the DFA accepting state, the input string *bbb* is accepted.

3.2.2 Tomita's algorithm in gtb

We can run Tomita's algorithm on an LR DFA using the method

```
this_derivation := tomita_1_parse[my_dfa STRING]
```

where, as for the LR parser, STRING is the input string, a doubly quoted string of grammar terminals.

The parser can be run on the LR(0), SLR(1), LALR or LR(1) DFA. The following script causes gtb to run the parser on ex8 with an LR(1) DFA.

```
(* ex8  GLR parsing *)
S ::=  A 'b' .
A ::=  'b' A | # .

(
ex8_grammar := grammar[S]
ex8_nfa := nfa[ex8_grammar lr 1]
render[open["nfa.vcg"] ex8_nfa]

ex8_dfa := dfa[ex8_nfa]
render[open["dfa.vcg"] ex8_dfa]

this_derivation := tomita_1_parse[ex8_dfa "bbb"]
render[open["gss_ex8.vcg"] this_derivation]
)
```

As a result of running the Tomita parser gtb produces a stack structured graph (ssg) that is a version of the GSS described above. Rendering this graph to a VCG file the stack structure can be viewed.

It is possible to get gtb to print the actions it performs during a traversal using gtb_verbose. Running the script

```
(* ex8  GLR parsing *)
S ::=  A 'b' .
A ::=  'b' A | # .

(
ex8_grammar := grammar[S]
```

```
     ex8_nfa := nfa[ex8_grammar lr 1]
     render[open["nfa.vcg"] ex8_nfa]
     ex8_dfa := dfa[ex8_nfa]
     render[open["dfa.vcg"] ex8_dfa]

     gtb_verbose := true
     this_derivation := tomita_1_parse[ex8_dfa "bbb"]
     render[open["ssg.vcg"] this_derivation]
     )
```

generates the output

```
******: Tomita 1 parse (queue length 0) : 'bbb'
Lexer initialised: lex_whitespace terminal suppresssed,
lex_whitespace_symbol_number 0
Lex: 3 'b'
State 22, input symbol 3 'b', action 7 (R[0] R11 |0|->4)
State 22, input symbol 3 'b', action 23 (S23)
State 24, input symbol 3 'b', action 27 (S27)
Lex: 3 'b'
State 23, input symbol 3 'b', action 7 (R[0] R11 |0|->4)
State 26, input symbol 3 'b', action 9 (R[2] R17 |2|->4)
State 23, input symbol 3 'b', action 23 (S23)
State 24, input symbol 3 'b', action 27 (S27)
Lex: 3 'b'
State 23, input symbol 3 'b', action 7 (R[0] R11 |0|->4)
State 26, input symbol 3 'b', action 9 (R[2] R17 |2|->4)
State 26, input symbol 3 'b', action 9 (R[2] R17 |2|->4)
State 23, input symbol 3 'b', action 23 (S23)
State 24, input symbol 3 'b', action 27 (S27)
Lex: EOS
State 27, input symbol 2 '$', action 10 (R[3] R18 |2|->5)
State 25, input symbol 2 '$', action 8 (R[1] R14 |1|->6 Accepting)
Lex: EOS
******: Tomita 1 parse: accept

SSG has final level 3 with 26 nodes and 26 edges; maximum queue length 1

Edge visit count histogram
0: 13
1: 12
2: 2
Total of 16 edge visits

Path length histogram
0: 3
4: 4
Total of 7 path entries
Weighted total of 16 path entries

Reduction length histogram
0: 3
4: 4
```

```
Total of 7 reduction length entries
Weighted total of 16 reduction length entries

Reduction histogram
0: 3
4: 4
Total of 7 reduction entries
Weighted total of 16 reduction entries
```

(Some discussion of the diagnostics produced from `tomita_1_parse` can be found in Section 3.3.1.)

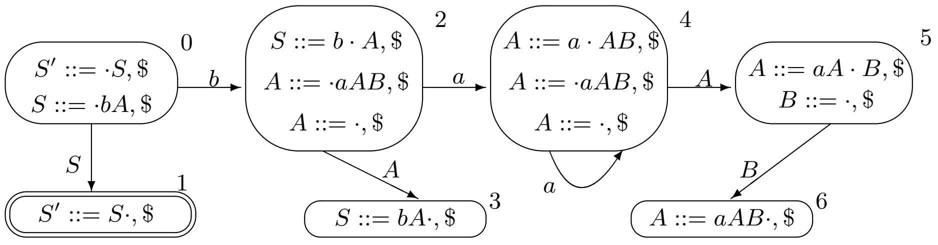
3.2.3 Right nullable rules

As we saw in Section 3.2.1, while the GSS is being constructed it is possible to add a new edge from an existing node, creating a new path down which a reduction must be applied. This was the case for node u_8 in the example in Section 3.2.1.

Tomita's algorithm was designed to construct the GSS as efficiently as possible, keeping to a minimum the amount of graph searching required. For this reason the worklist mentioned above is carefully designed so that pending reductions are stored with the first edge of each path down which they must be applied. The problem is that when a new edge is added to the middle of an existing path then reductions associated with nodes at the end of the path may not be applied down the new path. We illustrate the problem with the following grammar, ex9.

$$\begin{array}{lll} S & ::= & b\ A \\ A & ::= & a\ A\ B \mid \epsilon \\ B & ::= & \epsilon \end{array}$$

whose LR(1) DFA is



We run Tomita's algorithm with the above table and input string baa, using the `gtb` script

```
    (* ex9  Right nullable rules *)
    S ::=  'b' A .
    A ::=  'a' A B | # .
    B ::=   # .

    (
    ex9_grammar := grammar[S]
    ex9_nfa := nfa[ex9_grammar lr 1]
    render[open["nfa.vcg"] ex9_nfa]

    ex9_dfa := dfa[ex9_nfa]
    render[open["dfa.vcg"] ex9_dfa]

    this_derivation := tomita_1_parse[ex9_dfa "baa"]
    render[open["ssg.vcg"] this_derivation]
    )
```

This generates the GSS

The parser outputs diagnostics, in particular reporting that the input is rejected.

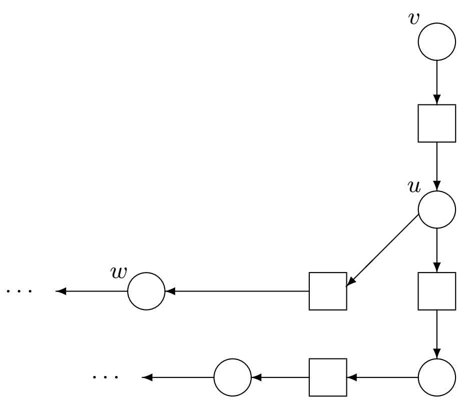
```
******: Tomita 1 parse (queue length 0) : 'baa'
******: Tomita 1 parse: reject

SSG has final level 3 with 12 nodes and 12 edges; maximum queue length 1

Edge visit count histogram
0: 7
1: 6
Total of 6 edge visits
```

The problem is that once node 34 in the GSS is constructed the reduction by the rule $A ::= aAB$ is applied and a new edge is created from node 33. This creates a new path from node 34 down which the reduction must be applied, but only reductions associated with node 33, and in this case there are none, are applied again. Thus the GSS construction process terminates without success.

In general, suppose that we add an edge from a node, u say, in the middle of an existing path in the GSS, and that there is a reduction associated with a node, v, further up this path.



It is not hard to see that the reduction must be of the form $A ::= \alpha\beta$ where $\beta \overset{*}{\Rightarrow} \epsilon$ and the item $A ::= \alpha \cdot \beta$ belongs to the DFA state that labels the node u. Rules of the form $A ::= \alpha\beta$ where $\beta \overset{*}{\Rightarrow} \epsilon$ and $\beta \neq \epsilon$ are called *right nullable rules.*

Tomita [Tom86] attempted to solve the problem of right nullable rules by introducing sub-levels into the GSS, but this caused the algorithm to fail to terminate in certain cases. Farshi [NF91] gave a different version of Tomita's algorithm, using a different worklist structure, and addressed the problem by simply searching the full GSS each time a new edge is created to ensure that all appropriate reductions are found and correctly applied. The problem with Farshi's algorithm is that it generates very much more graph searching during the GSS construction and hence the algorithm is much less efficient than Tomita's algorithm. Some statistics comparing the efficiency of the algorithms can be found in [JSE04].

In fact the problem can be solved by using Tomita's original algorithm but adding extra reductions to the LR parse table. We now describe the modification to the tables.

3.3 Right Nulled parse tables

As we remarked in Section 3.2.3, Tomita's algorithm does not always correctly parse an input string if the grammar contains right nullable rules. However, we notice that, for a rule of the form $A ::= \alpha\beta$, where $\beta \overset{*}{\Rightarrow} \epsilon$, if the parser reaches a state labelled with an item $(A ::= \alpha \cdot \beta, a)$ and if the next input symbol is a, then eventually, without reading any further input, the parser will reach a state labelled with $(A ::= \alpha\beta\cdot, a)$ and perform a reduction. Thus the parser could have performed the reduction from the state labelled $(A ::= \alpha \cdot \beta, a)$, popping off just the symbols associated with α from the stack. This simple observation forms the basis of the right nulled (RN) GLR parsers.

We construct the LR DFA for a grammar, LR(0), SLR(1), LALR or LR(1) as desired, exactly as for the standard LR parser, but states labelled with an item of the form $(A ::= \alpha \cdot \beta, a)$, where $\beta \overset{*}{\Rightarrow} \epsilon$ are also treated as reduction states. To perform a reduction we need to know the number of symbols to be popped off the stack and the name of the non-terminal on the left hand side of the rule. Thus, in an RN table we record reduction actions together with the number of symbols to be popped off the stack and the left hand side of the rule.

We construct an RN table from the LR DFA in a similar way to the LR table, putting sm and gm into the RN table exactly as for the LR(1) table. For a reduction, if the label of state h includes the item $(A ::= x_1 \dots x_p \cdot \beta, G)$, where $\beta \overset{*}{\Rightarrow} \epsilon$ and $A \neq S'$, put $r(p, A)$ in row h, column x for all $x \in G$. Finally, put acc into row k, column \$, where k is the DFA state whose label includes $(S' ::= S\cdot, \$)$, and if $S ::= \epsilon$ add acc to row 0, column \$, where 0 is the DFA start state.

3.3.1 RN tables in `gtb`

As we have said, `gtb` collects all the information that it needs to build an LR table when it constructs the NFA. Thus the method that builds an NFA contains a parameter which instructs `gtb` to treat slots of the form $A ::= \alpha \cdot \beta$, where $\beta \overset{*}{\Rightarrow} \epsilon$, as reductions. The default action is to use only the standard LR reductions. To include the right nullable reductions we use the method call

```
my_nfa := nfa[my_grammar lr 1 nullable_reductions]
```

If we run `gtb` with this NFA method and visualise the resulting NFA using VCG we see that the NFA states that contain a reduction slot are highlighted in blue. For the standard NFA it is the leaf nodes that are highlighted, but when the `nullable_reductions` option is used then other nodes may be highlighted as well.

For example, for the grammar ex9, running the script

```
(* ex9  Right nullable rules *)
S ::=  'b' A .
A ::=  'a' A B | # .
B ::=   # .

(
ex9_grammar := grammar[S]
rn_nfa := nfa[ex9_grammar lr 1 nullable_reductions]
render[open["nfa1.vcg"] rn_nfa]
)
```

and then running VCG on `nfa1.vcg` we see that the nodes labelled $S ::= a \cdot A$, $A ::= b \cdot AB$, and $A ::= bA \cdot B$, are highlighted.

We note that, for efficiency, the internal representation of an RN table in `gtb` is not simply the theoretical representation described above, it matches more closely the structure of the output table `parse.tbl` described in Section 2.2.3. The internal representation is not of immediate concern to the reader but, as for the LR parser, it is helpful for understanding some of the diagnostic information produced by `gtb`. In particular, when `gtb` reports on a reduction action it reports the `gtb` generated slot number of the item $A ::= x_1 \ldots x_p \cdot \beta$ (the number of the corresponding NFA node), then gives the length of the reduction and the associated non-terminal. In fact, as can be seen in Section 3.3.2, the diagnostics generated by the Tomita parser are in a similar form to those generated by the LR parser described above.

3.3.2 Tomita's algorithm with RN tables

We can use Tomita's algorithm with RN tables and the algorithm will work correctly on all grammars and input strings.

For example, running the script

```
(* ex9  Right nullable rules *)
S ::=  'b' A .
A ::=  'a' A B | # .
B ::=   # .

(
ex9_grammar := grammar[S]

rn_nfa := nfa[ex9_grammar lr 1 nullable_reductions]
```

```
    rn_dfa := dfa[rn_nfa]
    render[open["dfa1.vcg"] rn_dfa]
    write[open["parse.tbl"] rn_dfa]

    this_derivation := tomita_1_parse[rn_dfa "baa"]
    render[open["ssg1.vcg"] this_derivation]

    this_derivation := tomita_1_parse[rn_dfa "baab"]
    this_derivation := tomita_1_parse[rn_dfa "ca"]
    this_derivation := tomita_1_parse[rn_dfa "bbb"]
    )
```

generates the following GSS and DFA

The file `parse.tbl` produced is

```
LR symbol table
0 !Illegal
1 #
2 $
```

```
3 a
4 b
5 A
6 B
7 S
8 S!augmented

LR state table
28 0:
 S29
 S30
29 4:
 R[1] R[4] {2}
 S31
 S32
30 7:
 R[2] {2}
31 3:
 R[1] R[3] {2}
 S31
 S33
32 5:
 R[7] {2}
33 5:
 R[0] R[5] {2}
 S34
34 6:
 R[6] {2}

LR reduction table
R[0] R14 |0|->6 B ::= . #
R[1] R15 |0|->5 A ::= . #
R[2] R18 |1|->8 S!augmented ::= S . Accepting
R[3] R19 |1|->5 A ::= 'a' . A B
R[4] R20 |1|->7 S ::= 'b' . A
R[5] R21 |2|->5 A ::= 'a' A . B
R[6] R22 |3|->5 A ::= 'a' A B .
R[7] R23 |2|->7 S ::= 'b' A .
```

and running the script also produces the following diagnostics, showing that the first input string is correctly accepted and that the other three input strings are (correctly) rejected.

```
******: Tomita 1 parse (queue length 0) : 'baa'
******: Tomita 1 parse: accept

SSG has final level 3 with 16 nodes and 16 edges; maximum queue length 3

Edge visit count histogram
0: 3
1: 6
2: 6
3: 2
```

```
Total of 24 edge visits

******: Tomita 1 parse (queue length 0) : 'baab'
******: Tomita 1 parse: reject

SSG has final level 3 with 7 nodes and 6 edges; maximum queue length -1000

Edge visit count histogram
0: 7
Total of 0 edge visits

******: Tomita 1 parse (queue length 0) : 'ca'
Illegal lexical element detected

******: Tomita 1 parse (queue length 0) : 'bbb'
******: Tomita 1 parse: reject

SSG has final level 1 with 3 nodes and 2 edges; maximum queue length -1000

Edge visit count histogram
0: 3
Total of 0 edge visits
```

Notice that `gtb` also gives statistics relating to the Tomita parse of the input. In particular it reports the number of edge visits carried out during the construction of the GSS, in the first example there are 24. This allows the performance of Tomita's algorithm to be compared with other algorithms. To more detailed diagnostics as the parse proceeds we can switch on the verbose mode, as shown on page 52.

3.3.3 The RNGLR algorithm

We can construct an algorithm that is more efficient than Tomita's in the case of right nullable rules by observing that a reduction of the form $A ::= \alpha\beta\cdot$, where $\beta \overset{*}{\Rightarrow} \epsilon$ need not be applied when β matches ϵ, as the nulled reduction $A ::= \alpha \cdot \beta$ will generate the required portion of the GSS when it is applied.

It is not hard to see that a GSS edge between two nodes, u and v say, at the same level exits if and only if it was created as the result of applying a reduction $B ::= \gamma \cdot \delta$ with γ being matched to ϵ. In particular, a reduction $A ::= \alpha\beta\cdot$ is to be applied with β matching ϵ if and only if the first edges of the paths down which it is to be applied have their target nodes on the current level. (This was the case, for example, in ex9 where the first edge of the path down which $A ::= aAB$ was applied was from node 34 to node 33, both on level 3.) Thus to avoid unnecessarily applying $A ::= \alpha\beta\cdot$ we only apply reductions down paths whose first edge goes to a level below the current level. (It is important to note that, because of this modification, the RNGLR algorithm cannot be used with the standard LR tables if the grammar contains right nullable rules.)

The GSS constructed by the RNGLR algorithm does not have symbol nodes (however, when we draw them we label the GSS edges with symbols for pedagogic purposes). This makes the GSS smaller and more efficient to search.

It does have implications for derivation tree construction, the parser version of the algorithm constructs a GSS whose edges are labelled with tree nodes. However, this method results in slightly smaller trees and so we prefer it to Tomita's parser, which exploits the GSS symbol nodes.

The fact that the GSS does not include symbol nodes allows the RNGLR algorithm to have a more efficient pending reduction worklist, reductions are stored with the second edge on the path down which they are to be applied. This, together with the early application of right nullable reductions discussed at the start of this section means that the RNGLR algorithm is more efficient than Tomita's algorithm, even on grammars with no right nullable rules.

To run the RNGLR algorithm in `gtb` we use an method call of the form

```
    this_derivation := rnglr_recognise[my_dfa STRING]
```

For example, running the script

```
(* ex9  Right nullable rules *)
S ::=  'b' A .
A ::=  'a' A B | # .
B ::=   # .

(
ex9_grammar := grammar[S]
rn_nfa := nfa[ex9_grammar lr 1 nullable_reductions]
rn_dfa := dfa[ex9_nfa]

gtb_verbose := true
this_derivation := rnglr_recognise[rn_dfa "baa"]
render[open["ssg2.vcg"] this_derivation]
)
```

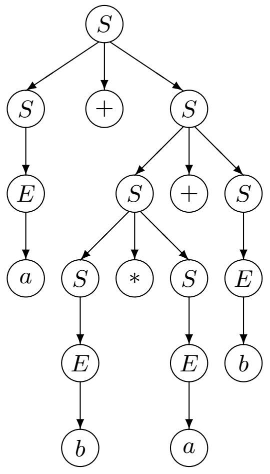
generates the following GSS

3.4 The RNGLR parser

Strictly speaking the algorithms that we have discussed so far are recognisers. A parser is a recogniser that outputs, in some form, a derivation of the input string, if that string is in the language.

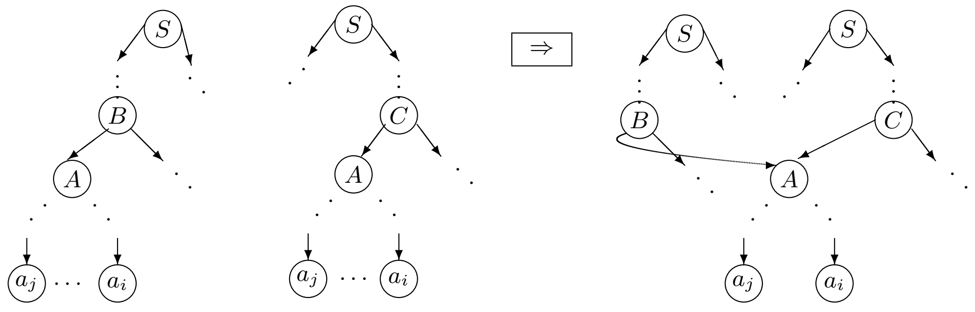
3.4.1 Shared packed parse forests

Derivations are often presented as trees. The root of a derivation tree is labelled with the start symbol, the leaves are labelled with the symbols of the input string and the interior nodes are labelled with nonterminals. The children of a node, A say, are labelled with the symbols from an alternate of the grammar rule for A. For example, the following derivation tree corresponds to the derivation on page 4.

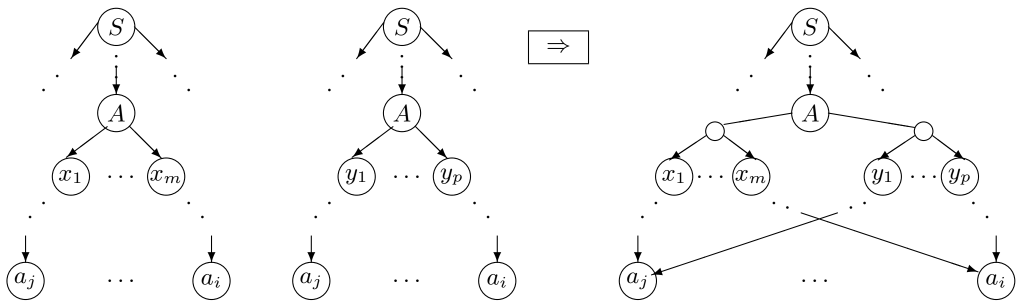


Tomita constructed the GLR algorithm with the production of derivation trees in mind. So the extension of the recogniser to a parser is relatively straightforward. However, GLR algorithms can be applied to all grammars, and ambiguous grammars have sentences that have more than one derivation tree. Thus we begin by describing an efficient representation of the set of derivation trees of a string.

We combine all the derivation trees for a string into a single structure called a shared packed parse forest in which common nodes are shared and multiple sets of children are packed together. In general, given a forest of derivation trees for a string $a_1 \dots a_n$, if two trees contain the same subtree for a substring $a_i \dots a_i$, say, then this subtree can be shared.

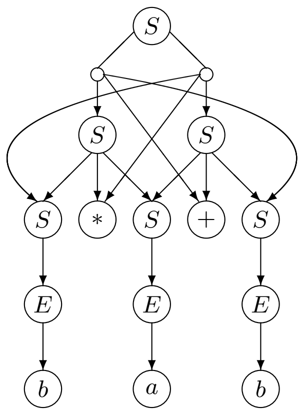


If two nodes labelled A, say, have different subtrees which derive the same substring, $a_j \dots a_i$ say, then the two nodes labelled A can be packed together and the subtrees can be added as alternates under the packed node.



A directed graph obtained by taking the derivation trees of a sentence u and merging and packing the nodes in the fashion described above is called a *shared packed parse forest* (SPPF).

The following is the SPPF for the (two) derivations of the string $b * a + b$ from the grammar ex1 on page 5



3.4.2 The RNGLR parser

To turn the RNGLR recogniser into a parser we construct an SPPF as the GSS is built. The method that we use is basically the same as that used by Rekers [Rek92] to turn Farshi's GLR algorithm into a parser. A node is constructed when an edge is added to the GSS, and the edge is labelled with this node.

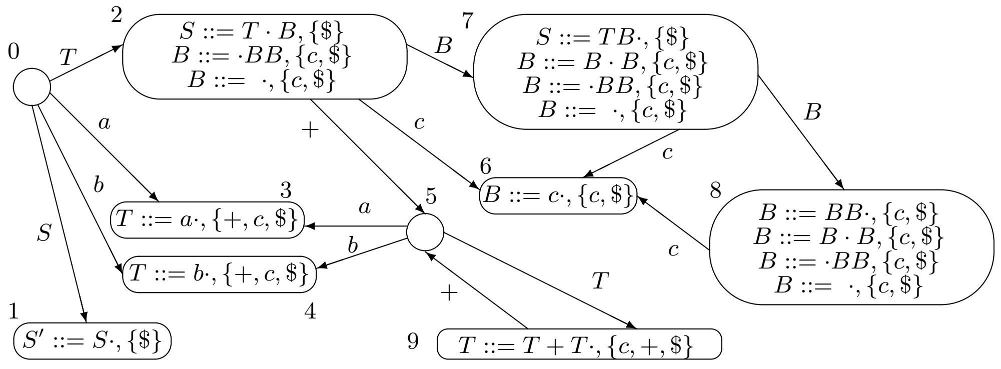
We have to treat right nullable rules carefully because in the RNGLR algorithm the right hand ends of such reductions are short circuited. Thus we have pre-constructed ϵ-SPPFs for the nullable right hand ends of rules and to add these in the appropriate places once the GSS construction has been completed. These SPPFs are passed into the algorithm along with the parse table and the input string.

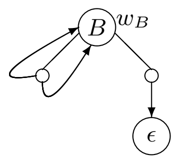
The SPPF is then constructed as follows. When an input symbol, a say, is read at the ith step of the algorithm, an SPPF node, u say, labelled (a, i) is created. All of the GSS edges created when this a is read are labelled u. When a reduction $A ::= \alpha \cdot \beta$ is applied the labels, $u_1, \ldots, u_k$ say, on the edges on the path down which the reduction is applied are collected. If the last node on the path is at level j then we look for an SPPF node labelled (A, j). If one has already been constructed at this step then we create a new packing node as a child of this node. Otherwise we create an SPPF node labelled (A, j). The newly created node is then given as children the nodes labelled $u_k, \dots, u_1$. If $\beta \neq \epsilon$ then the root node of the SPPF for β is also made a child of the new SPPF node. (For full details of the SPPF construction process see [SJ06] or [SJH00].)

For example, consider the grammar ex10

$$\begin{array}{lcl} S & ::= & T\ B \\ T & ::= & T\ +\ T \mid a \mid b \\ B & ::= & B\ B \mid c \mid \epsilon \end{array}$$

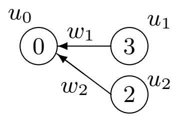
which has the following RN SLR(1) DFA and ϵ-SPPF

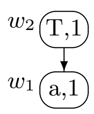




We parse the string $a + b + a$ as follows.

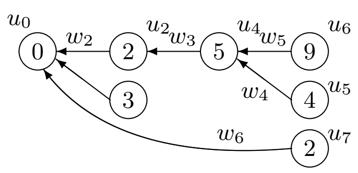
We create a GSS node u_0 labelled 0 and then read the first input symbol, a. We create an SPPF node, w_1, labelled $(a, 1)$, a GSS node, u_1, labelled 3 and an edge (u_1, u_0) labelled w_1. We then perform the reduction $T ::= a$ down this edge. We create an SPPF node, w_2 labelled $(T, 1)$ as a parent of w_1, a GSS node u_2 labelled 2 and an edge (u_2, u_0) labelled w_2.

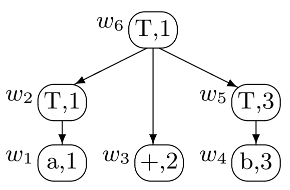




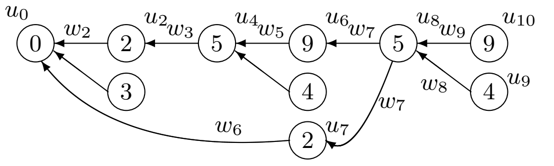
We read the next input symbol, $+$, create an SPPF node, w_3, labelled $(+, 2)$, a GSS node, u_4, labelled 5 and an edge (u_4, u_2) labelled w_3.

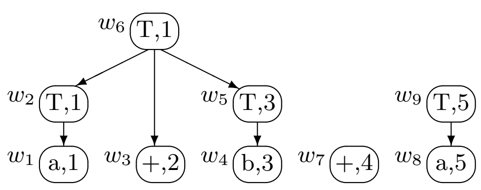
Reading the next input symbol, b, we create an SPPF node, w_4, labelled $(b, 3)$, a GSS node, u_5, labelled 4 and an edge (u_5, u_4) labelled w_4. We then perform the reduction $T ::= b$ down the edge (u_5, u_4), create an SPPF node, w_5 labelled $(T, 3)$, a GSS node u_6 labelled 9 and an edge (u_6, u_4) labelled w_5. Performing the reduction from u_6 we trace back along the path of length 3, collecting the labels w_5, w_3 and w_2. We create a new SPPF node, w_6, labelled $(T, 1)$ with children w_2, w_3, w_5, a GSS node, u_7, labelled 2 and an edge (u_7, u_0) labelled w_6.



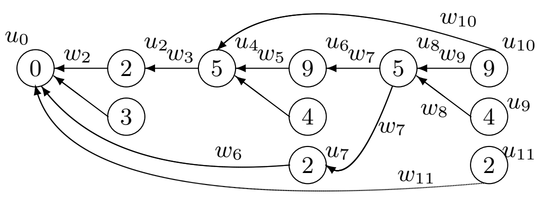


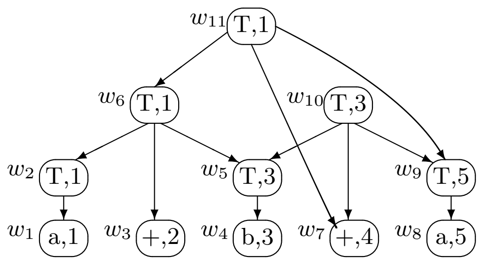
We then read the next symbol, $+$, create an SPPF node, w_7, labelled $(+, 4)$, a GSS node, u_8, labelled 5, and two edges (u_8, u_6) and (u_8, u_7) labelled w_7. Then we read the final symbol, a, create an SPPF node, w_8, labelled $(a, 5)$, a GSS node, u_9, labelled 5 and an edge (u_9, u_8) labelled w_8. Applying the reduction $T ::= a$ down the edge (u_9, u_8), we create an SPPF node, w_9 labelled $(T, 5)$, a GSS node u_{10} labelled 9 and an edge (u_{10}, u_8) labelled w_9.



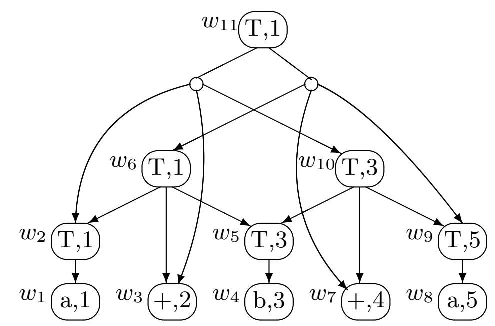


From u_{10} we trace back along the two paths of length 3. This results in the GSS and SPPF shown below.



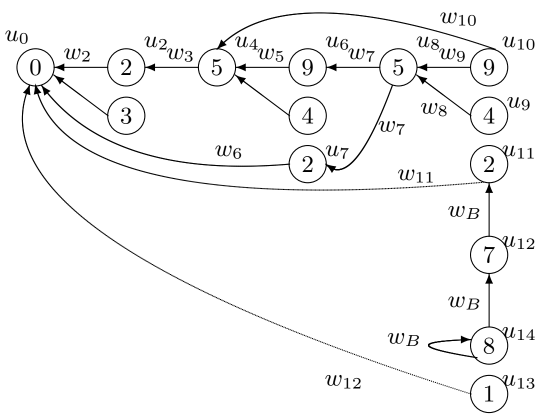


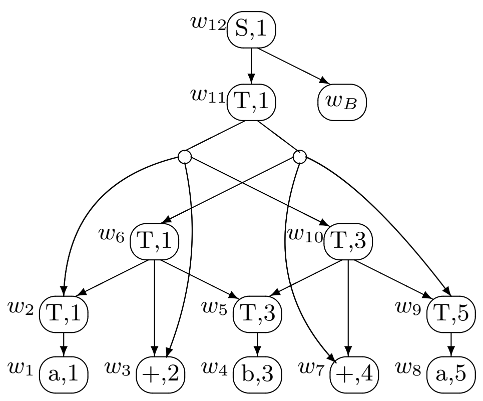
We have added a new edge to node u_{10} so the reduction is applied down this edge. The required GSS node and edge already exist, and there is already an SPPF node labelled $(T, 1)$ that we have constructed at this step. Thus we reuse this node and add the second set of children using packing nodes.



We then apply the reduction $B ::= \epsilon$ from node u_{11}, creating a new GSS node, u_{12}, labelled 7 and an edge (u_{12}, u_{11}) labelled w_B. Applying the reduction $B ::= \cdot BB$ does not create any new edges. Applying the reduction $S ::= T \cdot B$ from u_{11} we create an SPPF node labelled $(S, 1)$ with children w_{11} and w_B, and a GSS node, u_{13} labelled 1.

We then apply the reduction $B ::= \epsilon$ from u_{12} to create u_{14} labelled 8. As the edges from u_{12} and u_{14} were created by a zero length reduction we do not apply any of the non-zero length reductions in state 7 or 8. We apply the reduction $B ::= \epsilon$ to u_{14}, creating an edge labelled w_B from this node to itself, and the construction process is complete.





To run the parser version of the RNGLR algorithm in `gtb` we use the method

```
this_derivation := rnglr_parse[my_dfa STRING]
```

Chapter 4

Reduction incorporated recognisers

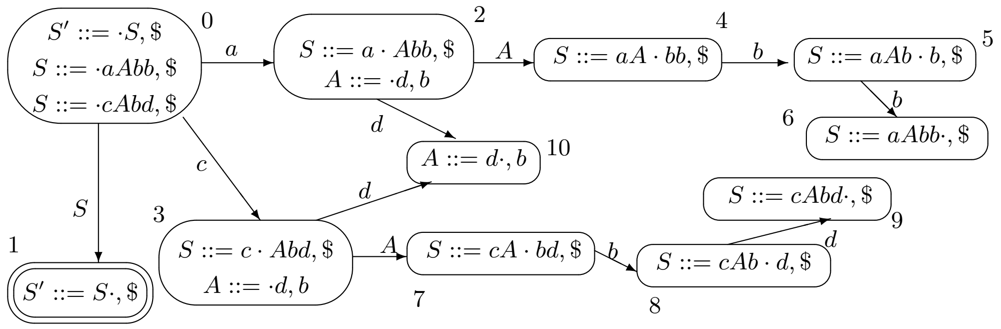
Although they are relatively efficient, GLR algorithms are at least cubic order in worst case. There has been quite a lot of research directed towards improving the efficiency of the standard LR parsing algorithm by reducing the cost of the stack activity. We know that there exist context-free languages that cannot be recognised by a finite state automaton, and thus we cannot expect to remove the stack completely from the GLR algorithm. However, from a theoretical point of view, we only require a stack to deal with instances of self embedding, i.e. derivations of the form $A\overset{*}{\Rightarrow}\alpha A\beta$ where α and β are not ϵ.

This observation forms the basis of a different type of general parsing algorithm, initially developed by Aycock and Horspool [AH99]. As this is a tutorial manual we shall not give the detailed motivation for this approach in terms of the standard LR algorithm, the interested reader can read about this in [SJ02] or [SJ05]. However, the basic idea can be illustrated as follows.

We use a stack in the standard LR parser so that when we perform a reduction we can find the state that we need to trace back to. For example, consider the grammar, ex11,

$$\begin{array}{lcl} S & ::= & a\ A\ b\ b \mid c\ A\ b\ d \\ A & ::= & d \end{array}$$

which has LR(1) DFA



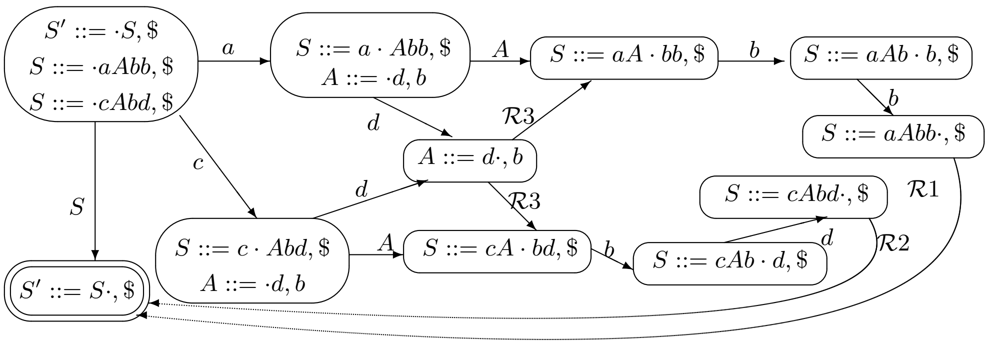
On input *adbb* we eventually read all the input and have stack

$$0 \leftarrow 2 \leftarrow 4 \leftarrow 5 \leftarrow 6$$

Since $S ::= aAbb\cdot$ is in state 6, we trace back to state 0 by popping four symbols off the stack and then we traverse the S-transition from state 0 to state 1.

What happens if we extend the LR DFA by adding special reduction transitions that move directly from a reduction state to the state that should be pushed on to the stack? So, in the above example we would just have a reduction transition (a special form of ϵ-transition that does not require reading an input symbol) from state 6 to state 1. This would mean that we did not have to push and pop the intermediate states.

In fact we could put a reduction transition from state 6 to state 1, and from state 9 to state 1, without changing the language accepted by the resulting PDA. However, there is a problem with the reduction associated with state 10. There are two possible paths to state 10 and two corresponding states that may need to be traced back to, states 4 and 7.



We could put reduction transitions in for each of these, but when the automaton is in state 10 it would not be possible to tell which of the two reduction transitions should be taken. The above automaton will incorrectly accept the strings *cabb* and *abbd*.

The solution is to 'multiply out' the nodes, creating one copy of node 10 for each of the possible paths. We now discuss the details of this approach.

4.1 Grammars without self-embedding

A grammar has *self embedding* if there is some non-terminal, A, and strings $\alpha, \beta \neq \epsilon$ such that $A \stackrel{*}{\Rightarrow} \alpha A \beta$.

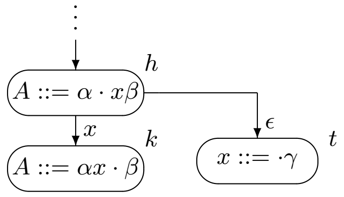
In this section we discuss the construction of a finite state automaton, with special reduction transitions, which accepts precisely the language of a grammar provided that the grammar does not contain any self embedding.

4.1.1 Reduction incorporated automata

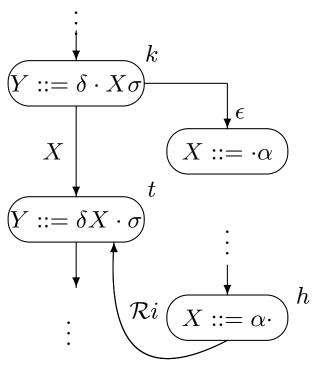
We begin not with the LR DFA but with the LR NFA, and add reduction transitions from leaves of the tree. Thus, initially we construct an intermediate automaton, IRIA(Γ), that is similar to the LR(0) NFA except that grammar slots can label more that one node, and the header nodes are omitted.

Initially we suppose that the grammar does not contain any recursion.

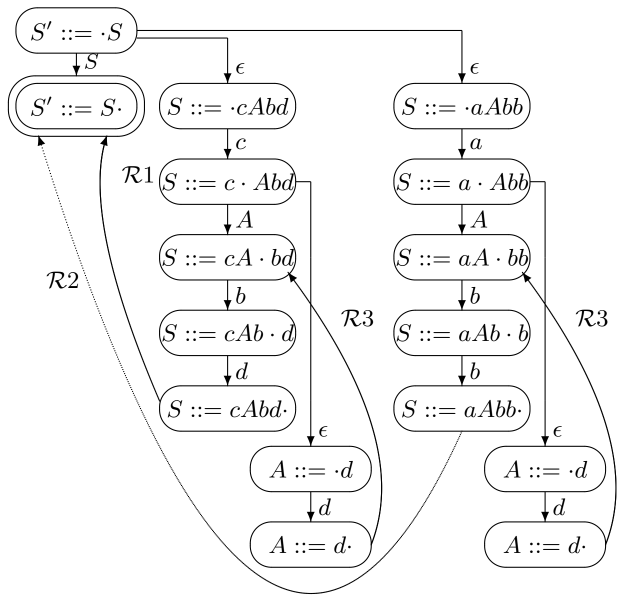
We begin by constructing a node labelled $S' ::= S\cdot$, this is the start node. While the graph has leaf nodes labelled $A ::= \alpha \cdot \beta$ where $\beta \neq \epsilon$, pick such a leaf node, h say, and suppose that $\beta = x\beta'$. Create a new node, k say, labelled $A ::= \alpha x \cdot \beta'$ and a transition from h to k labelled x. If x is a non-terminal, then for each rule $x ::= \gamma$ create a new node, t say, labelled $x ::= \cdot\gamma$ and an ϵ-transition from h to t.



Once all the leaf nodes have labels of the form $X ::= \alpha\cdot$, we add the reduction transitions. For each state, h, labelled $X ::= \alpha\cdot$, where $X ::= \alpha$ is rule i, trace back up the automaton until the first node, k say, with a label of the form $Y ::= \delta \cdot X\sigma$ is reached. If t is the state such that there is a transition labelled X from k to t, add a transition labelled $\mathcal{R}i$ from h to t.



This approach results in the following FA, IRIA(Γ_{11}), for ex11 above.

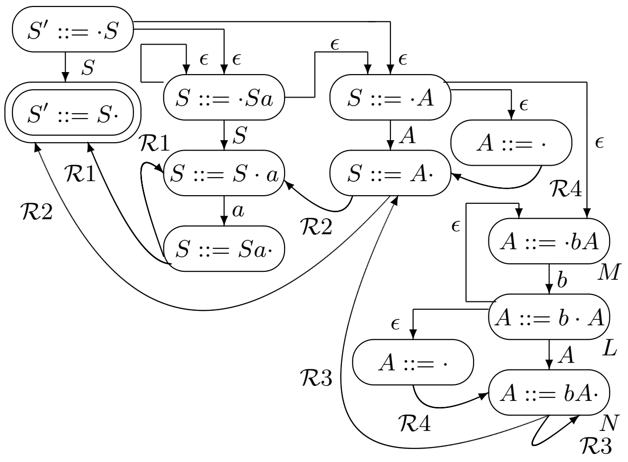


In the case of recursive rules we cannot simply use the multiplying out approach because this would never terminate. So, for recursive instances of non-terminals we add an ϵ-edge back to the most recent instance of the target item on a path from the start state to the current state.

For example, from the recursive grammar ex12 given by

1. S	$::=$	Sa		3. A	$::=$	bA
2. S	$::=$	A		4. A	$::=$	ϵ

we generate the IRIA, IRIA(Γ_{12})



The ϵ-edge from L to M and the corresponding $\mathcal{R}3$-edge from N to itself arise from the recursive occurrence of A in the rule $A ::= bA$.

For completeness we give the formal IRIA construction algorithm.

IRIA construction algorithm

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.
   (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 arrow 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 $X ::= x_1 \dots x_n\cdot$ 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 $(\epsilon, x_1, \dots, x_n)$ 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.

Step5: Mark the node labelled $S' ::= \cdot S$ as the start node and mark the node labelled $S' ::= S\cdot$ as the accepting node.

A string is accepted by an IRIA if it is accepted by the automaton in the standard way when the $\mathcal{R}i$ transitions are treated as ϵ-transitions.

The following theorem is proved in [SJ05].

Theorem 1 *Let Γ be a CFG that does not contain any self embedding. Then a string, u, of terminals is accepted by IRIA(Γ) if and only if u is in $L(\Gamma)$.*

4.1.2 IRIA(Γ) in `gtb`

In `gtb` the construction of the 'multiplied out' version of an LR NFA is referred to as *unrolling*. To instruct `gtb` to build the IRIA for a grammar, Γ, we use the method

```
my_iria := nfa[my_grammar unrolled 0]
```

The parameter `lr` is replaced with the parameter `unrolled`. As for LR NFAs, the `nfa` method augments the grammar, if it is not already in augmented form, before building the IRIA.

Although the formal version of an IRIA does not include header nodes, the `gtb` representation does use header nodes, because this allows a particularly efficient internal representation. The VCG rendering of the IRIA includes the header nodes.

The script

```
(* ex11  IRIA construction *)
S ::=  'a' A 'b' 'b' | 'c' A 'b' 'd' .
A ::=  'd' .

(
ex11_grammar := grammar[S]
ex11_iria:= nfa[ex11_grammar unrolled 0]
render[open["nfa.vcg"] ex11_iria]
)
```

creates the IRIA

In the VCG graph the ϵ-transitions are red and the reduction transitions are blue. An ϵ-transition that returns to an existing node because of recursion is green.

The script

```
(* ex12 *)
S ::=  S 'a' | A .
A ::=  'b' A | # .

(
ex12_grammar := grammar[S]
ex12_iria:= nfa[ex12_grammar unrolled 0]
render[open["nfa.vcg"] ex12_iria]
)
```

creates the IRIA

Note: `gtb` will construct IRIA(Γ) for any context-free grammar Γ. However, IRIA(Γ) only correctly accepts $L(\Gamma)$ if Γ does not contain any self embedding.

4.1.3 Reduction incorporated automata

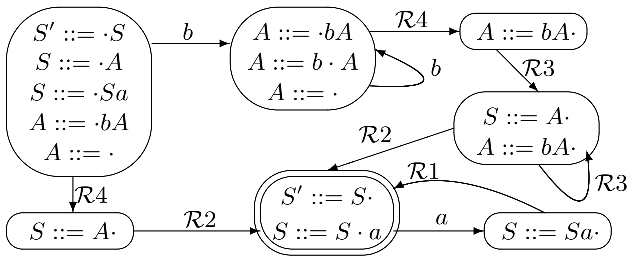
The RIGLR algorithm that we shall discuss below is able to correctly determine whether or not a string of terminals is accepted by any given reduction incorporated automaton. However, the algorithm is more efficient if there is less non-determinism in the automaton.

The FAs IRIA(Γ) that we constructed above are highly non-deterministic. In this section we shall consider approaches that reduce the non-determinism in IRIA(Γ), culminating in the definition of a *reduction incorporated automaton* for Γ.

First we note that IRIA(Γ) only ever has input which is a string of terminals, thus the transitions labelled with non-terminals can be removed once the $\mathcal{R}$-transitions have been constructed. So from now on we shall assume that IRIA(Γ) has had all the non-terminal labelled transitions removed.

A non-deterministic automaton can always be transformed in to a deterministic one by applying the standard subset construction. In IRIA(Γ) the transitions labelled $\mathcal{R}$ consume no input and so can be treated as ϵ-transitions from the point of view of the subset construction. This is fine if all we want is a recogniser but ultimately we want to produce all the derivations of a sentence u and thus we do not want to lose the information about which reductions were used.

We remove from IRIA(Γ) the transitions labelled with non-terminals and then apply the subset construction treating the $\mathcal{R}$-transitions as non-ϵ transitions. The following automaton is the result of applying this process to IRIA(Γ_{12}).



As there are no transitions to the start state of IRIA(Γ), the start state of RIA(Γ) is the unique state whose label includes the item $S' ::= \cdot S$. The accepting states are the states whose label includes the item $S' ::= S\cdot$. Note that an RIA can have more than one accepting state.

To construct an RIA using `gtb` we apply the subset construction to the corresponding IRIA. The required `gtb` method call is

```
my_ria := dfa[my_iria]
```

Remember RIA(Γ) only accepts precisely $L(\Gamma)$ if Γ does not contain any proper self embedding. In the next section we will describe how to deal with grammars which do contain proper self embedding.

4.2 Recursion call automata

In this section we describe how to build a push down automaton, RCA(Γ), for any given context-free grammar, Γ, that can be used to recognise sentences in $L(\Gamma)$.

We begin by modifying the grammar to remove most of the recursion by creating 'terminalised' instances, $A^{\perp}$, of recursive non-terminals (this changes the language generated by the grammar). We then construct the RIA for the modified grammar as described in the previous sections. We also construct an RIA for each terminalised non-terminal, and then call these automata from the main RIA. The automata calls are managed using a stack and so the resulting structure is a (non-deterministic) push down automaton.

4.2.1 Terminalising a grammar

Given a grammar, Γ, our goal is to replace instances of non-terminals on the right hand sides of rules with special terminals, until the resulting grammar has no self embedding.

For example, given the grammar ex13

$$\begin{array}{lcl} S & ::= & A\ a \\ A & ::= & a\ B \mid a \\ B & ::= & A\ c \end{array}$$

we have that $A \overset{*}{\Rightarrow} aAc$. If we replace the rule $B ::= Ac$ with the rule $B ::= A^{\perp}c$, and treat $A^{\perp}$ as a terminal, then the derivation is no longer possible. The grammar

$$\begin{array}{lcl} S & ::= & A\ a \\ A & ::= & a\ B \mid a \\ B & ::= & A^{\perp}\ c \end{array}$$

does not contain self embedding. Notice, this is not the only terminalisation that we could have used. The following grammar also has no self embedding

$$\begin{array}{lcl} S & ::= & A\ a \\ A & ::= & a\ B^{\perp} \mid a \\ B & ::= & A\ c \end{array}$$

We shall call a grammar that has, possibly, terminalised versions of some of its non-terminals a *terminalised grammar*. (Note that it is possible to terminalise any instance of any non-terminal, even if there is no self embedding involved. We may wish to do this because it can reduce the size of the final RCA.)

Detecting self embedding in a grammar and determining which non-terminals to terminalise is not completely trivial. The GDG, see Section 1.6, can be used to assist this process as we shall discuss in Section 4.2.4. For now it is sufficient to say that we can construct an RCA from any terminalised context-free grammar that does not have self embedding, using a process that we shall now describe.

4.2.2 RCA(Γ)

Given a grammar, Γ, terminalise instances of non-terminals, as required, and so that the resulting grammar, Γ_S say, has no self embedding. We shall call Γ_S a *derived grammar*.

For each non-terminal, $A \neq S$, such that Γ_S contains at least one terminalised instance of A, construct the grammar Γ_A that has the same rules as Γ_S but start symbol A.

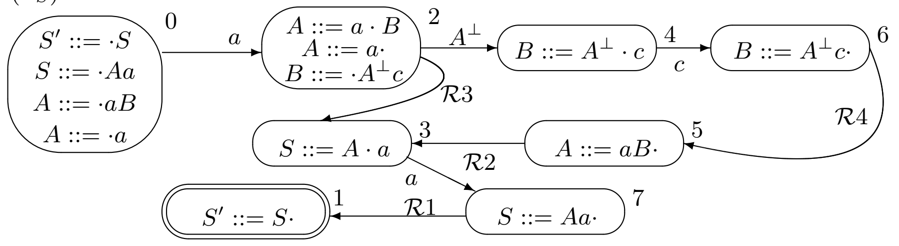
Construct RIA(Γ_S) and construct RIA(Γ_A) for each terminalised non-terminal A, ensuring that all the states in all the automata have different numbers.

For example, for the terminalised grammar, Γ_S,

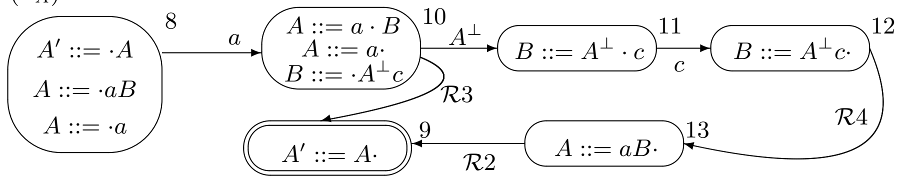
$$\begin{array}{lcl} S & ::= & A\ a \\ A & ::= & a\ B \mid a \\ B & ::= & A^{\perp}\ c \end{array}$$

we have the following RIAs,

RIA(Γ_S)



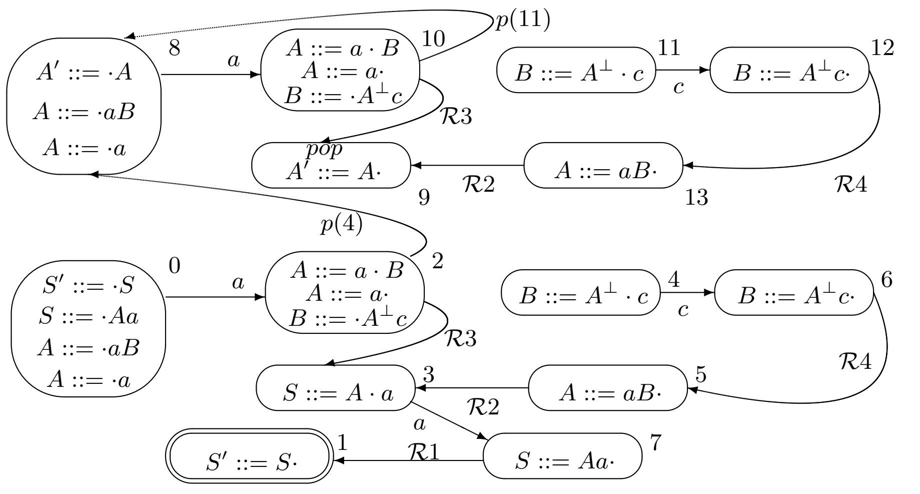
RIA(Γ_A)



To combine the RIAs into a push down automaton, RCA(Γ), that recognises $L(\Gamma)$ we replace transitions labelled with a terminalised non-terminal, $A^{\perp}$, with a call to the corresponding RIA, RIA(Γ_A). When RIA(Γ_A) has matched an appropriate part of the input then we need to return to the next state in the calling automaton, and continue to match the rest of the string. Thus we replace each transition labelled $A^{\perp}$ with a transition to the start state of RIA(Γ_A) and push the target of the $A^{\perp}$-transition on to a stack. Thus we label the new transition $p(k)$, where k is the target of the $A^{\perp}$-transition. These new transitions are ϵ-transitions in the sense that they do not consume any of the input string, but they have an associated stack action. The accepting states of the automata RIA(Γ_A) are labelled as pop states in the RCA. (If Γ_S contains $S^{\perp}$ then the accepting state of the RCA is also a pop state.)

We call such a PDA a *recursion call automaton* (RCA).

The PDA, RCA(Γ_{13}) constructed from the derived grammar Γ_S of the grammar ex13 above is



4.2.3 Traversing an RCA

We traverse an RCA with an input string u as follows. We begin in the start state with an empty stack and read the first input symbol. At each step in the traversal we will be in a current state, h say, with a current stack and have read the current input symbol, x say.

- ◇ If h has a transition labelled x to state k, say, then we may choose to move to state k and read the next input symbol.
- ◇ If h has a transition labelled $\mathcal{R}i$ to state k, say, then we may choose to move to state k, without reading the next input symbol.
- ◇ If h has a transition labelled $p(t)$ to a state k, say, then we may choose to push t on to the stack and move to state k, without reading the next input symbol.
- ◇ If h is labelled as a pop state then we may choose to pop the top symbol, t say, off the stack and move to state t, without reading the next input symbol.
- ◇ If h is the RCA accepting state and x is the end of string symbol, \$, then we may choose to terminate the traversal and report success.
- ◇ If none of the above actions is possible then we terminate the traversal and report failure.

Of course it is possible for the RCA to be non-deterministic, i.e. that more than one of the above actions is possible at some step. Thus we need an algorithm that computes all possible traversals on a given input string, the equivalent of Tomita's algorithm for LR automata. We shall discuss such an algorithm in Section 4.3, but first we describe the construction of RCAs in `gtb`.

4.2.4 Constructing RCAs in gtb

We have already discussed, in Sections 4.1.2 and 4.1.3, the generation of RIAs in `gtb`. To construct an RCA for a grammar Γ, `gtb` needs to construct the desired terminalised grammar, Γ_S, from which it can then build the required RIAs.

There are two possible approaches. The user can specify the instances of non-terminals that are to be terminalised to construct Γ_S, or `gtb` can identify instances of self embedding and automatically introduce terminalisations until the resulting grammar does not contain self embedding. In this section we describe the former approach. In a later section we shall discuss the automatic generation of Γ_S.

`gtb` allows the user to mark instances of non-terminals in a grammar as ‘to be replaced with a terminalised version’ for an RIGLR parser. We use `~` to mark the non-terminals.

For example, `gtb` accepts the script

```
(* ex13  terminalising a grammar *)
S ::=  A 'a' .
A ::=  'a' B | 'a' .
B ::= ~A 'c' .

(
ex13_grammar := grammar[S]
)
```

The method `grammar[S]` ignores the `~` annotations and treats an instance of `~A` as though it were just `A`. Thus all of the previously described behaviour of `gtb` is unchanged by the inclusion of the `~` annotations. This is to make it possible to run the GLR algorithms on exactly the same input grammars as the RIGLR algorithms.

We can terminalise a grammar, replacing each instance `~A` with a pseudo terminal called `A!tilde` by `gtb`, using the method

```
terminalise_grammar[my_grammar]
```

In order for `gtb` to use the terminalisation notation the `grammar` method needs to have the `tilde_enabled` option set.

```
grammar[start_symbol tilde_enabled]
```

This ensures that `gtb` includes the terminalised non-terminals in the list of grammar symbols when the grammar is built. Running the script

```
(* ex13  terminalising a grammar *)
S ::=  A 'a' .
A ::=  'a' B | 'a' .
B ::= ~A 'c' .

(
    ex13_grammar := grammar[S tilde_enabled]
    terminalise_grammar[ex13_grammar terminal]
    write[ex13_grammar]

    iria_S := nfa[ex13_grammar unrolled 0]
    render[open["nfa.vcg"] iria_S]
    )
```

builds the IRIA

and causes the following to be printed on the screen

```
Terminalising A
```

```
Grammar report for start rule S
Grammar alphabet
   0 !Illegal
   1 #
   2 $
   3 'A!tilde'
   4 'B!tilde'
   5 'S!tilde'
   6 'a'
   7 'c'
-------------
   8 A
   9 B
  10 S

Grammar rules
A ::= 'a' B[0] |
        'a' .
B ::= ~'A!tilde' 'c' .
S ::= A[0] 'a' .

End of grammar report for start rule S
```

The original grammar, `grammar_S` has been mutated into a new grammar in which one of the instances of the non-terminal A has been replaced by a new terminal `A!tilde`.

Since the grammar as been mutated the original unterminalised grammar is lost. To reconstruct it, to mutate the grammar back to its original form, we use the `nonterminal` option with the `terminalise_grammar` function.

```
my_grammar := terminalise_grammar[my_grammar nonterminal]
```

The corresponding option to terminalise a grammar is `terminal` and this is the default.

4.3 The RIGLR algorithm

The RIGLR algorithm takes as input an RCA for a grammar Γ and a string $u = a_1 \ldots a_d$ and it traverses the RCA using u. The algorithm accepts u if it is a sentence in Γ and rejects u otherwise.

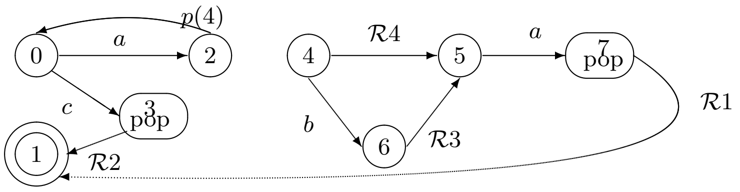
The algorithm starts in the RCA start state and at each step i it constructs the set U of all RCA states that can be reached on reading the input $a_1 \ldots a_i$. When a push transition is traversed the return state is pushed onto the stack. The stacks are combined into a graph structured stack called the *call graph* and each state in U is recorded with the node in the call graph that corresponds to the top of the associated stack.

We use a slightly modified version of the algorithm that incorporates some lookahead. The lookahead is similar to that used in the SLR(1) version of the GLR algorithms. We only perform a push action to the automaton for A if the next input symbol is in $\textsc{first}(A)$, or $\textsc{follow}(A)$ if $A \overset{*}{\Rightarrow} \epsilon$. Also, we only traverse a reduction transition if the next input symbol is in FOLLOW(A) where A is the left hand side of the reduction rule, and we only perform a pop action from the automaton for A if the next input symbol is in FOLLOW(A).

For example, consider the simple grammar ex14

$$\begin{array}{lcl} S & ::= & a\ S^{\perp}\ B\ a \mid c \\ B & ::= & b \mid \epsilon \end{array}$$

which has RCA

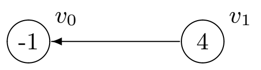


and consider the input string *aacbaa*.

We begin by creating a base node, v_0, in the call graph labelled -1 and start in the start state 0. Thus we have

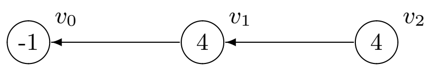
$$U = \{(0, v_0)\}$$

The only action is to read the next input symbol, a, and move to state 2, starting the next step. We then perform the push action, create a new call graph node, v_1 labelled 4, and move to state 0.



$U = \{(2, v_0),\ \ (0, v_1)\}$

Reading the next input symbol, a, from the process $(0, v_1)$ we move to state 2, and from here we perform the push action, creating a new call graph node v_2.



$U = \{(2, v_1),\ \ (0, v_2)\}$

We then read the next input symbol, c, and from the process $(0, v_2)$ we move to state 3. From state 3 we traverse the reduction transition to state 1 and we perform the pop action. In the latter case we pop the top element, 4 the label of v_2, and we move to state 4. The top of the stack is now the child of v_2, v_1. We do not traverse the reduction transition from state 5 because the next input symbol, b, is not in FOLLOW(B).

$$U = \{(3, v_2),\ \ (1, v_2),\ \ (4, v_1)\}$$

The next input symbol is b and from state 4 we can move to state 6 and then, via the reduction transition, to state 5. From $(5, v_1)$, reading the next input symbol a, we move to state 7. We can traverse the reduction transition to state 1 and perform the pop action, creating the process $(4, v_0)$, and then the reduction transition from state 4 to state 5.

$$U = \{(7, v_1),\ \ (1, v_1),\ \ (4, v_0),\ \ (5, v_0)\}$$

Reading the last input symbol, a, we move from state 5 to state 7. We can traverse the reduction to state 1 but, as v_0 is the base of the stack, no pop action can be performed.

$$U = \{(7, v_0), \quad (1, v_0)\}$$

The traversal is now complete and one of the current positions, $(1, v_0)$, is the accepting state and the empty stack. Thus the input string is (correctly) accepted.

To run the RIGLR algorithm in `gtb` we use the method

```
ri_recognise[my_grammar STRING]
```

This method takes a grammar with terminalisation annotation, generates the structures needed for an RIGLR recogniser as described above, and then runs the recogniser on the specified STRING.

For example, running the script

```
(* ex14 the RIGLR algorithm *)
S ::=  'a' ~S B 'a' | 'c' .
B ::=  'b' | # .

(
ex14_grammar := grammar[S tilde_enabled]
ri_recognise[ex14_grammar "aacbaa"]
)
```

generates the following output.

```
Terminalising S

******: RIGLR recognise: 'aacbaa'
******: RIGLR recognise: accept
Call graph has 3 nodes and 2 edges
```

4.4 Terminalising a grammar

We now return to the issue of terminalising a grammar so that all the self embedding is removed. In general there will be different terminalisation possibilities that we could choose. We call a set of instances of non-terminals which have been replaced with pseudo-terminals to remove self embedding a *terminalisation* of the grammar. A terminalisation is *minimal* if no proper subset of it is also a terminalisation.

The problem that has to be addressed is how to identify minimal terminalisations. The GDG constructed by `gtb` has been designed to facilitate this process. Recall from Section 1.6 that the GDG for a grammar, Γ, is a graph with nodes labelled with each of the non-terminals of Γ such that there is an edge from A to B if B appears on the right hand side of the grammar rule for A.

The edge from A to B is labelled R (for non-trivial right context) if there is a rule $A ::= \alpha B \beta$ such that $\beta \neq \epsilon$ and the edge is labelled L (for non-trivial left context) if there is a rule $A ::= \sigma B \tau$ such that $\sigma \neq \epsilon$.

A *path of length k* in a graph is a sequence of nodes $(N_1, \ldots, N_{k+1})$ such that there is an edge from N_i to N_{i+1}, for $1 \leq i \leq k$, and a *cycle* (from N_1 to itself) is a path $(N_1, \ldots, N_k, N_1)$, of length at least 1, such that $N_i = N_j$ if and only if $i = j$. Then a non-terminal A is recursive if and only if there is a cycle in the GDG from A to itself.

The non-terminal A is self embedding if and only if there is a path in the GDG from A to itself that contains at least one edge labelled L and at least one edge labelled R. We call such paths *LR-paths.* Similarly, we call a path that contains at least one L (R) edge an L-(R-)path. (So every LR-path is also an L-path and an R-path.) To determine whether a non-terminal A is self embedding we can find all the edges in the GDG that lie on any path from A to itself and look at their labels. Finding all such edges is called *strongly connected component analysis.*

4.4.1 Strongly connected components

A graph is said to be *strongly connected* if there is a path from any node to any other node in the graph.

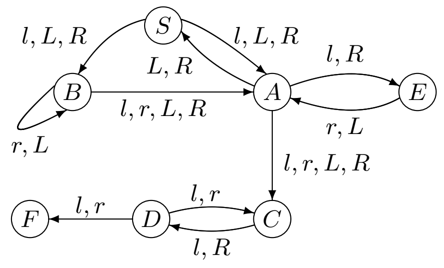
For any graph and any node A in the graph we can form the strongly connected component containing A. We take A and all the nodes that are on any path from A to itself. We then form a subgraph by taking these nodes and all the edges from the original graph between these nodes.

Formally we partition the set of nodes of a graph into subsets, two nodes A and B are in the same subset if and only if there is path from A to B and a path from B to A. We turn the each set in the partition in to a graph by adding an edge from A to B if A and B are in the same partition and if there was an edge from A to B in the original graph. The subgraphs constructed in this way are all strongly connected and they are called the *strongly connected components* (SCCs) of the graph.

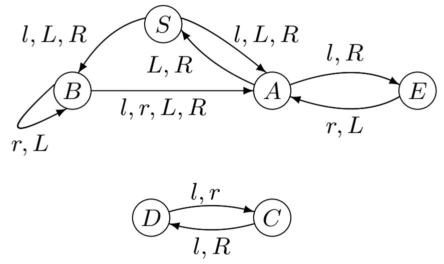
For example, the grammar ex15

$$\begin{array}{lcl}
S & ::= & A\ B\ A\ a \mid B\ d \\
A & ::= & E\ a \mid b \mid C\ S\ C \mid a\ S\ b \mid \epsilon \\
B & ::= & b\ a\ A\ B \mid A \\
C & ::= & D\ a \mid a \\
D & ::= & C \mid F \\
E & ::= & d\ A \\
F & ::= & a
\end{array}$$

has GDG



which in turn has two strongly connected components.



In order to remove recursion, we need to identify cycles in the SCCs and then remove an edge from each cycle, by terminalising the corresponding instance(s) of the non-terminal which is the target of the edge.

4.4.2 Finding terminalisation sets

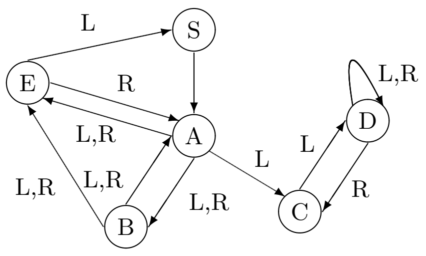
Any LR-path from a node to itself in a GDG is contained in a maximal SCC, and thus we begin by using Tarjan's algorithm, as above, to find the SCCs.

Each SCC that contains at least one edge labelled L and at least one edge labelled R is then considered. In any terminalisation all of the L-cycles or all of the R-cycles must have been removed, and once all of the L-cycles (or R-cycles) have been removed there can be no remaining self embedding. Thus to find precisely all the minimal terminalisations we run the process twice, once to find all possible minimal terminalisations that can be obtained by removing edges from every L-cycle and then again to find the minimal terminalisations by removing edges from every R-cycle. (Of course, the two processes will find many of the same terminalisations.) In the rest of this discussion we shall describe the process of generating terminalisations by removing L-cycles, the process for R-cycles is identical except that only R-cycles are considered.

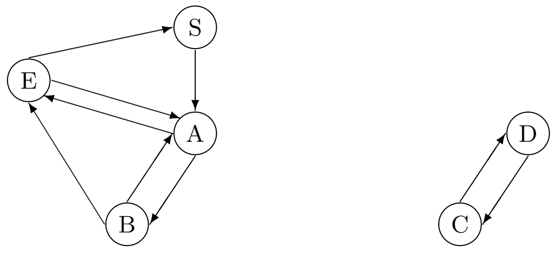
We illustrate the process using the following grammar, ex16, as a running example.

$$\begin{array}{lll@{\qquad\qquad}lll}
S & ::= & A & C & ::= & a\ D \\
A & ::= & E\ E\ B\ C & D & ::= & C\ D\ a \mid \epsilon \\
B & ::= & a\ A\ E\ a \mid \epsilon & E & ::= & A\ S \mid \epsilon
\end{array}$$

The grammar has GDG

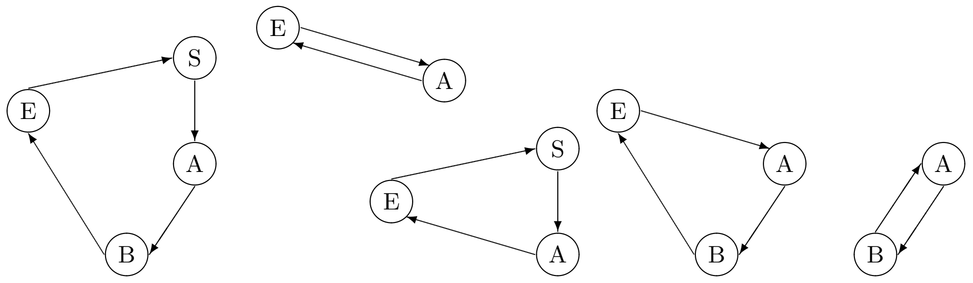


First we run Tarjan's algorithm. Since we are removing L-cycles, all L-loops (L-edges from a node to itself) must be included in any terminalisation and no other loops are of interest. To reduce the size of the SCC's we remove all loops. For the above example this gives the following two LR-SCC's.



We then consider each LR-SCC in turn.

We begin by finding all the cycles, and then we consider all the L-cycles. For the first SCC in our example this results in five L-cycles.



Once all the cycles have been found, the basic algorithm works recursively down the list of cycles choosing one edge from each cycle to form each terminalisation set in turn. Of course this approach will produce many terminalisation sets that are not minimal as well as all the minimal ones. As a first step towards efficiency, as each cycle is considered the algorithm checks to see if the terminalisation set already contains an edge from this cycle, and if it does then the cycle is skipped.

For example, with the above cycles we could choose (S, A) from the first cycle, (A, E) from the second cycle, then nothing more from the third cycle because we already have (S, A), then (A, B) from the fourth cycle and nothing from the fifth cycle. This gives us the following terminalisation set

$$\{(S, A) \quad (A, E) \quad (A, B)\}$$

We then back track to the fourth cycle and choose (B, E) instead of (A, B). This time we have to choose an edge from the fifth cycle, thus we get two terminalisation sets

$$\{(S,A)\quad (A,E)\quad (B,E)\quad (A,B)\} \qquad\qquad \{(S,A)\quad (A,E)\quad (B,E)\quad (B,A)\}$$

Carrying on in this way we find that the algorithm constructs the following sequence of terminalisation sets, from which repeated ones have been removed. (The actual list of terminalisations constructed before the test for subsets will depend on the order in which the cycles are visited.)

{(S,A) (A,E) (E,A) (A,B)}
{(S,A) (E,A) (A,B)}
{(A,B) (A,E)}
{(A,B) (E,A) (E,S)}
{(B,E) (A,E) (B,A)}
{(B,E) (E,A) (S,A) (B,A)}
{(B,E) (E,A) (A,E) (B,A)}
{(B,E) (E,A) (E,S) (B,A)}
{(E,S) (A,E) (B,E) (B,A)}
{(E,S) (A,E) (E,A) (B,A)}
{(E,S) (E,A) (B,A)}

{(S,A) (A,E) (E,A) (B,A)}
{(S,A) (E,A) (B,A)}
{(A,B) (E,A) (A,E)}
{(B,E) (A,E) (A,B)}
{(B,E) (E,A) (S,A) (A,B)}
{(B,E) (E,A) (A,E) (A,B)}
{(B,E) (E,A) (E,S) (A,B)}
{(E,S) (A,E) (B,E) (A,B)}
{(E,S) (A,E) (E,A) (A,B)}
{(E,S) (E,A) (A,B)}

We then remove the non-minimal sets by removing any set which has one of the other sets as a proper subset.

For our running example this results in the following six sets.

{(S,A) (E,A) (A,B)}
{(S,A) (E,A) (B,A)}
{(A,B) (A,E)}
{(B,E) (A,E) (B,A)}
{(E,S) (E,A) (A,B)}
{(E,S) (E,A) (B,A)}

We then repeat the process for the R-cycles. Once the terminalisations for R-cycles are constructed we run a final test to see whether any of the L-terminalisations are subsets of the R-terminalisations, or vice versa. Then the resulting sets are the minimal terminalisations for the SCC.

In our example the R-cycles are the same as the L-cycles so no additional terminalisation sets are created.

We then compute the terminalisation sets for the other SCC's, and combine them to form terminalisation sets for the whole grammar.

In our example the other SCC has two minimal terminalisations

$$\{(C,D),(D,D)\} \qquad\qquad \{(D,C),(D,D)\}$$

4.4.3 Terminalising a grammar using `gtb`

`gtb` supports user-defined grammar terminalisation by performing GDG analysis that identifies the strongly connected components. The terminalisation analysis is performed by issuing the method call

```
cycle_break_sets[my_gdg]
```

It uses Tarjan's algorithm [Tar72] on the initial GDG and, as a side effect, `gtb` creates a new version of the GDG in which the nodes in each equivalence class and the edges between them are identified, the nodes in equivalence classes of greater than one element are all coloured the same colour, and the edges between nodes in the same equivalence class have class 1 or 3. The class 2 edges can be hidden in VCG allowing the non-trivial strongly connected components to be viewed directly. In addition, the set of minimal terminalisation sets for each SCC is output to the screen. The edges in the sets are listed by number and these numbers are displayed in the `gdg.vcg` file.

Consider the following example, ex16,

```
(* ex16  removing self embedding *)
S ::= A .
A ::= E E B C .
B ::= 'a' A E 'a' | # .
C ::= 'a' D .
D ::= C D 'a' | # .
E ::= A S | # .

(
ex16_grammar := grammar[S]
ex16_gdg := gdg[ex16_grammar]

cycle_break_sets[ex16_gdg]
render[open["gdg.vcg"] ex16_gdg]
)
```

When this script is run the following output is generated. (More detailed information can be obtained by setting `gtb_verbose` to true.)

```
Break sets for partition 1
0: {6, 8} cardinality 2
1: {7, 8} cardinality 2

Break sets for partition 2
0: {1, 2} cardinality 2
2: {2, 9, 10} cardinality 3
3: {2, 9, 11} cardinality 3
5: {1, 4, 5} cardinality 3
10: {4, 9, 10} cardinality 3
11: {4, 9, 11} cardinality 3
```

After the analysis the GDG looks similar to the original GDG but nodes S, B, A and E are coloured the same colour as each other, and so are nodes C and D. Within the VCG tool there is a 'Folding' option which allows nodes and edges to be combined and hidden. (In the Windows version of VCG there is a Folding option on the task bar.) Under Folding is an option to Expose/Hide edges. If this option is selected and used to Hide the class 2 edges then the following graph is displayed

Choosing the terminalisation set $\{6,8\} \cup \{2,3\}$, we use the GDG to find the corresponding slots in the grammar and introduce the associated terminalisation to the grammar.

```
(* ex16 *)
S ::= A .
A ::= ~E ~E ~B C .
B ::= 'a' A E 'a' | # .
C ::= 'a' ~D .
D ::= C ~D 'a' | # .
E ::= A S | # .

(
ex16_grammar := grammar[S]
terminalise_grammar[ex16_grammar terminal]

ex16_gdg := gdg[ex16_grammar]
cycle_break_sets[ex16_gdg]
render[open["gdg.vcg"] ex16_gdg]
)
```

Running this script we see that the terminalised grammar has no non-trivial strongly connected components, and hence no self embedding.

4.4.4 Pruning the search space

In some cases the number of interim terminalisation sets constructed prior to the minimality testing is very large. Furthermore, for some grammars the number of cycles, and the final number of minimal terminalisation sets, means that the prospect of finding all minimal terminalisation sets is impractical.

We can get `gtb` to list the interim terminalisation sets by stopping the analysis before the minimality testing step. We do this using the `retain_break_sets` option.

```
cycle_break_sets[my_gdg retain_break_sets]
```

(The option to do the minimality testing is `prune_break_sets_by_table`, which is the default option.)

Running the script

```
(* ex16 *)
S ::= A .
A ::= E E B C .
B ::= 'a' A E 'a' | # .
C ::= 'a' D .
D ::= C D 'a' | # .
E ::= A S | # .

(
ex16_grammar := grammar[S]
ex16_gdg := gdg[ex16_grammar]

cycle_break_sets[ex16_gdg retain_break_sets]
)
```

generates the following output

```
Break sets for partition 1
0: {6, 8} cardinality 2
1: {7, 8} cardinality 2
```

```
Break sets for partition 2
0: {1, 2} cardinality 2
1: {1, 2, 9} cardinality 3
2: {2, 9, 10} cardinality 3
3: {2, 9, 11} cardinality 3
4: {1, 2, 4} cardinality 3
5: {1, 4, 5} cardinality 3
6: {1, 2, 4, 9} cardinality 4
7: {1, 4, 5, 9} cardinality 4
8: {1, 4, 9, 10} cardinality 4
9: {1, 4, 9, 11} cardinality 4
10: {4, 9, 10} cardinality 3
11: {4, 9, 11} cardinality 3
```

In the case where the number of interim sets is very large we can reduce the search space by only looking for sets of size less than some specified value N say. The effect of this is that `gtb` stops attempting to construct each terminalisation set at the point where the $(N + 1)$st element is about to be added. We achieve this using a third parameter to the analysis function

```
cycle_break_sets[my_gdg retain_break_sets 2]
```

(To compute all the terminalisation sets we set the parameter to 0, which is the default option.)

For example, running the script

```
(* ex16 *)
S ::= A .
A ::= E E B C .
B ::= 'a' A E 'a' | # .
C ::= 'a' D .
D ::= C D 'a' | # .
E ::= A S | # .

(
ex16_grammar := grammar[S]
ex16_gdg := gdg[ex16_grammar]

cycle_break_sets[ex16_gdg retain_break_sets 2]
)
```

generates the following output

```
Break sets for partition 1
0: {6, 8} cardinality 2
1: {7, 8} cardinality 2

Break sets for partition 2
0: {1, 2} cardinality 2
Break set cardinality limit of 2 triggered: terminating
```

4.5 Aycock and Horspool's approach

The RIGLR algorithm is based on work done by Aycock and Horspool [AH99]. However, Aycock and Horspool's construction is slightly different. They construct an automaton based on a trie constructed from the 'handles' of a terminalised version of the input grammar. This method requires the grammar to have terminalised so that all recursion except for non-hidden left recursion has been removed. Aycock and Horspool also give a different algorithm for computing all the traversals of an automaton. However, their algorithm is only guaranteed to terminate if the original grammar did not contain any hidden left recursion. Of course, the RIGLR algorithm can be used on Aycock and Horspool style automata, and the RIGLR will work correctly on all context free grammars. To allow Aycock and Horspool's automata and algorithm to be studied and compared to other algorithms, `gtb` supports the construction of Aycock and Horspool trie based automata.

In this section we shall describe the trie based automata and their construction using `gtb`.

4.5.1 Left contexts and prefix grammars

The standard LR(0) parser identifies strings of the form $\alpha\beta$, where β is the right hand side of a grammar rule and there is a right-most derivation $S\underset{rm}{\Rightarrow}\alpha\beta w$, for some string of terminals w. The strings β are often called *handles* and the string α is called a *left context* of β. An LR(0) parser identifies a handle, and having identified a handle replaces it with the left hand side of the corresponding grammar rule. The set of strings $\alpha\beta$ as described above is the language accepted by the LR(0) DFA of the grammar.

Aycock and Horspool's automaton is constructed by taking all the strings in the language of the LR(0) DFA and forming a *trie* from them. When the trie has been constructed, reduction transitions are added, the non-terminal transitions are removed, and transitions labelled with terminalised non-terminals are replaced with calls to a trie constructed from that non-terminal.

In order to build the trie it is necessary for the language of the LR(0) DFA to be finite, and this is the case if and only if the grammar contains no recursion other than non-hidden left recursion.

To construct the language of an LR(0) DFA, we use what we call a *prefix grammar*. We augment the original grammar with a non-terminal S' and then for each non-terminal A, including S', we create a corresponding non-terminal, $[A]$ in the prefix grammar. The terminals of the prefix grammar are the terminals and non-terminals of the original grammar. The rules of the prefix grammar are constructed as follows. There is always a rule

$$[S'] ::= \epsilon$$

and for each instance of a non-terminal, B say, on the right hand side of a rule

$A ::= \gamma B \delta$ in the original grammar, where A is reachable, [1] there is a rule

$$[B] ::= [A]\gamma$$

in the prefix grammar. The prefix grammar, $P\Gamma$, has the property that $L([A])$, the language generated by the non-terminal $[A]$, is precisely the set of left contexts of A. I.e. the set of α such that $S \Rightarrow \alpha A w$ for some string of terminals w.

For example, consider the following terminalised version of ex15, from which we have also removed the right recursion in B.

$$\begin{array}{lcl} S & ::= & A\ B\ A\ a \mid B\ d \\ A & ::= & E^{\perp}\ a \mid b \mid C\ S^{\perp}\ C \mid a\ S^{\perp}\ b \mid \epsilon \\ B & ::= & b\ a\ A\ B^{\perp} \mid A \\ C & ::= & D^{\perp}\ a \mid a \\ D & ::= & C \mid F \\ E & ::= & d\ A \\ F & ::= & a \end{array}$$

(Notice that the non-terminals D, E and F are unreachable in this grammar, but they are used later to construct a recogniser for the original grammar.) The rules of the corresponding prefix grammar are

$$\begin{array}{lcl} [S'] & ::= & \epsilon \\ [S] & ::= & [S'] \\ [A] & ::= & [S] \mid [S]\ A\ B \mid [B]\ b\ a \mid [B] \\ [B] & ::= & [S]\ A \mid [S] \\ [C] & ::= & [A] \mid [A]\ C\ S^{\perp} \end{array}$$

If $\alpha\beta$ is in the language of the LR(0) DFA, where β is a handle, then there is some non-terminal, A say, such that $S \Rightarrow \alpha A w \Rightarrow \alpha\beta w$ and so $\alpha \in L([A])$. Thus the language of the LR(0) DFA is the set of all strings

$$\{\alpha\beta \mid \text{for some nonterminal } A, \alpha \in L([A]) \text{ and } A ::= \beta\}$$

For the above example we have

$$\begin{array}{ll} L([S']) & = \{\epsilon\}, \\ L([S]) & = \{\epsilon\}, \\ L([A]) & = \{\epsilon,\ AB,\ Aba,\ ba,\ A\}, \\ L([B]) & = \{A,\ \epsilon\}, \\ L([C]) & = \{\epsilon,\ AB,\ Aba,\ ba,\ A,\ CS^{\perp},\ ABCS^{\perp},\ AbaCS^{\perp},\ baCS^{\perp},\ ACS^{\perp}\} \end{array}$$

and the language of the LR(0) DFA for this grammar is

$$\begin{array}{ll} \{ & S,\ ABAa,\ Bd,\ E^{\perp}a,\ b,\ CS^{\perp}C,\ aS^{\perp}b,\ \epsilon,\ ABE^{\perp}a,\ ABb,\ ABCS^{\perp}C, \\ & ABaS^{\perp}b,\ AB,\ AbaE^{\perp}a,\ Abab,\ AbaCS^{\perp}C,\ AbaaS^{\perp}b,\ Aba,\ baE^{\perp}a, \\ & bab,\ baCS^{\perp}C,\ baaS^{\perp}b,\ ba,\ AE^{\perp}a,\ Ab,\ ACS^{\perp}C,\ AaS^{\perp}b,\ A,\ baAB^{\perp}, \\ & AA,\ baAB^{\perp},\ A,\ D^{\perp}a,\ a,\ ABD^{\perp}a, ABa,\ CS^{\perp}D^{\perp}a,\ CS^{\perp}a, \\ & ABCS^{\perp}D^{\perp}a,\ ABCS^{\perp}a,\ AbaD^{\perp}a,\ Abba,\ baD^{\perp}a,\ baa\ AD^{\perp}a,\ Aa, \\ & AbaCS^{\perp}D^{\perp}a,\ AbaCS^{\perp}a,\ baCS^{\perp}D^{\perp}a,\ baCS^{\perp}a,\ ACS^{\perp}D^{\perp}a,\ ACS^{\perp}a\} \end{array}$$

[1] A symbol x is reachable if $S \overset{*}{\Rightarrow} \tau x \sigma$ for some τ and σ.

gtb will automatically construct the prefix grammar as the first step in the trie construction discussed below. However, it is possible to get gtb to build the prefix grammar independently of the other methods using the method

```
prefix_grammar[my_grammar]
```

This method does not mutate the grammar, it creates a new grammar, in this case written to prefix_grammar.

For example, running the script

```
(* ex15 *)
S ::=  A B A 'a' | B 'd' .
A ::=  ~E 'a' | 'b' | C ~S C | 'a' ~S 'b' | # .
B ::=  'b' 'a' A ~B | A .
C ::=  ~D 'a' | 'a' .
D ::=  C | F .
E ::=  'd' A .
F ::=  'a' .

(
ex15_grammar := grammar[S tilde_enabled]
augment_grammar[ex15_grammar]
ex15_prefix := prefix_grammar[ex15_grammar]
write[ex15_prefix]
)
```

generates the following output.

```
Terminalising E
Terminalising S
Terminalising S
Terminalising B
Terminalising D

Grammar report for start rule S!left_context
Grammar alphabet
   0 !Illegal
   1 #
   2 $
   3 'A!terminal'
   4 'B!terminal'
   5 'C!terminal'
   6 'S!tilde'
   7 'a'
   8 'b'
   9 'd'
-------------
  10 A!left_context
  11 B!left_context
  12 C!left_context
  13 D!left_context
  14 E!left_context
  15 F!left_context
  16 S!augmented!left_context
  17 S!left_context

Grammar rules
A!left_context ::= B!left_context[0] 'b' 'a' |
                    B!left_context[0] |
                    E!left_context[0] 'd' |
                    S!left_context[0] |
                    S!left_context[0] 'A!terminal' 'B!terminal' .
B!left_context ::= S!left_context[0] 'A!terminal' |
                    S!left_context[0] .
C!left_context ::= A!left_context[0] |
                    A!left_context[0] 'C!terminal' 'S!tilde' |
                    D!left_context[0] .
D!left_context ::= UNDEFINED
E!left_context ::= UNDEFINED
F!left_context ::= D!left_context[0] .
S!augmented!left_context ::= UNDEFINED
S!left_context ::= S!augmented!left_context[0] .

End of grammar report for start rule S!left_context
```

4.5.2 Trie based automata

If Γ has no recursion other than non-hidden left recursion then all the sets $L([A])$ are finite. We form a trie from the set of strings in the language of the DFA as follows.

Form the set, $\Phi(\Gamma)$, of triples

$$\Phi(\Gamma) = \{(\alpha, \beta, A) \mid \text{for some nonterminal } A, \alpha \in L([A]) \text{ and } A ::= \beta\}$$

by first generating each of the sets $L([A])$ then, for each $\alpha \in L([A])$ and for each rule $A ::= \beta$ add (α, β, A) to $\Phi(\Gamma)$. (So the language of the DFA is the set of strings $\alpha\beta$ such that $(\alpha, \beta, A) \in \Phi(\Gamma)$.)

We use $\Phi(\Gamma)$ to construct a trie based automaton. Create a start node, u_0 say. For each element (α, β, A) in $\Phi(\Gamma)$, begin at the start node and suppose that x_1 is the first element of $\alpha\beta$. If there is an edge labelled x_1 from the start node move to the target of this edge. Otherwise create a new node, u say, and an edge labelled x_1 from u_0 to u and move to u. Continue in this way, so suppose that we are at node v and that the next symbol of $\alpha\beta$ is x_i. If there is an edge labelled x_i from v move to the target of this edge. Otherwise create a new node, w say, and an edge labelled x_i from v to w and move to w.

When all the symbols in $\alpha\beta$ have been read, so we are at a node, y say, that is at the end of a path labelled $\alpha\beta$ from u_0, if $A \neq S'$ retrace back up the path labelled with the elements of β, so that we are at a node, t say, that is the end of a path labelled α from u_0. If there is not an edge labelled A from t then create a new node, r say, and an edge from t to r labelled A. Add an edge labelled $\mathcal{R}$ from y to r.

We then remove the edges labelled with non-terminals from the original grammar, obtaining an automaton that corresponds to RIA(Γ_S) in the RIGLR algorithm. We call this the trie based automaton for S.

For example consider the terminalised grammar ex17

$$\begin{array}{lcl} S & ::= & a\ S^{\perp}\ b \mid A\ a\ A \mid S\ a \\ A & ::= & a\ B^{\perp} \mid a\ B^{\perp}\ B^{\perp} \mid \epsilon \\ B & ::= & B\ A \mid \epsilon \end{array}$$

The prefix grammar is

$$\begin{array}{lcl} [S'] & ::= & \epsilon \\ [S] & ::= & [S'] \mid [S] \\ [A] & ::= & [S] \mid [S]\ A\ a \end{array}$$

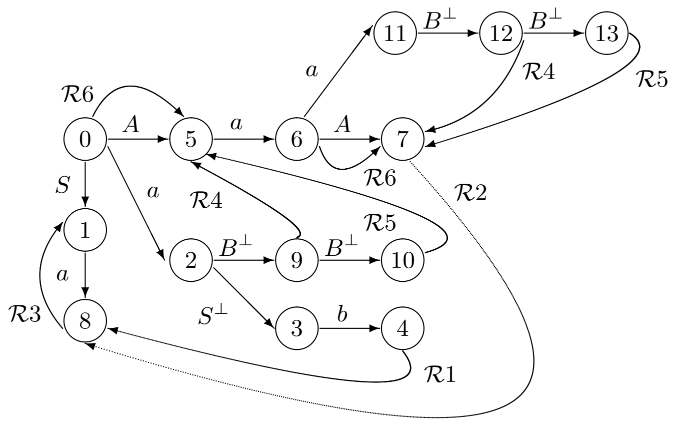
the languages are

$$L([S']) = \{\epsilon\}, \quad L([S]) = \{\epsilon\}, \quad L([A]) = \{\epsilon,\ Aa\}$$

and the set $\Phi(\Gamma)$ is

$$\{\ (\epsilon, S, S'),\ (\epsilon, aS^{\perp}b, S),\ (\epsilon, AaA, S),\ (\epsilon, Sa, S),\ (\epsilon, aB^{\perp}, A),\ (\epsilon, aB^{\perp}B^{\perp}, A),\ (\epsilon, \epsilon, A),\ (Aa, aB^{\perp}, A),\ (Aa, aB^{\perp}B^{\perp}, A),\ (Aa, \epsilon, A)\ \}$$

The corresponding trie based automaton for S (before the non-terminal transitions are removed) is



To build the full push down automaton, for each terminalised non-terminal, A, in turn we create a new rule $A' ::= A$ in the terminalised grammar and consider the grammar obtained by taking A' as the start symbol. We construct a prefix grammar and a trie based automaton for A, as we have described above for S'.

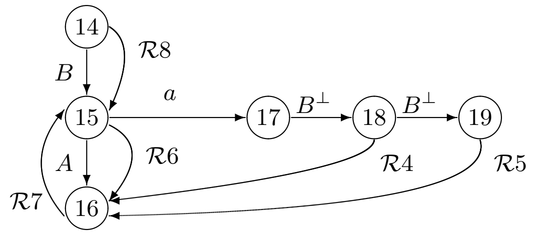
For the above example we need to build the trie based automaton for B. The required prefix grammar is

$$\begin{array}{lcl} [B'] & ::= & \epsilon \\ [B] & ::= & [B'] \mid [B] \\ [A] & ::= & [B]\ B \end{array}$$

the set $\Phi(\Gamma_B)$ is

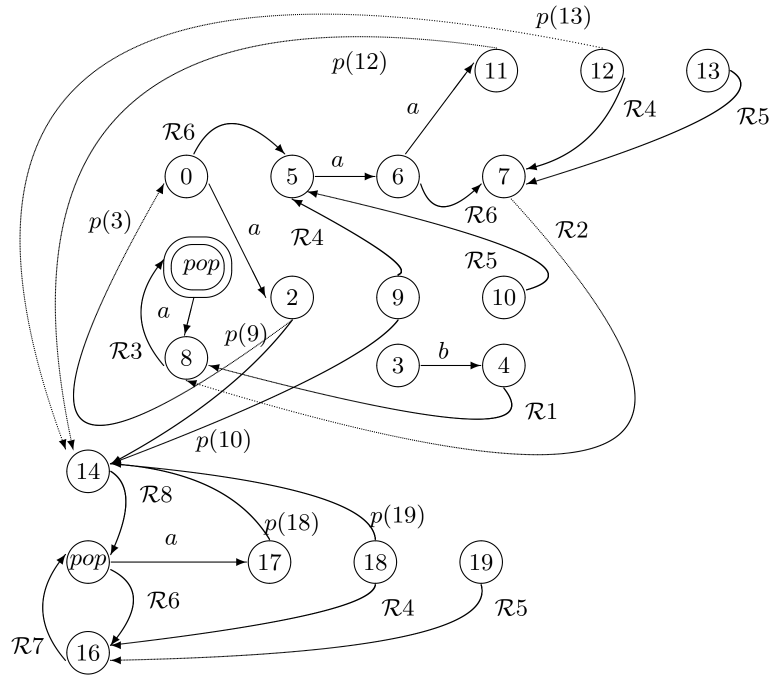
$$\{(\epsilon, B, B'),\ (\epsilon, BA, B),\ (\epsilon, \epsilon, B),\ (B, aB^{\perp}, A),\ (B, aB^{\perp}B^{\perp}, A),\ (B, \epsilon, A)\ \}$$

and the trie based automaton is



As for the RCA construction described in Section 4.2, we build a push down automaton from the trie based automata. For each transition in any of the automata labelled $A^{\perp}$, replace this with a transition labelled $p(k)$ to the start state of the trie based automaton for A, where k is the target of the $A^{\perp}$ transition. The accepting state of the PDA is the state in the trie based automaton for S which is the target of the S transition from the start state. For each terminalised non-terminal, $A^{\perp}$, the state in the trie based automaton for A which is the target of the A transition from the start state is a *pop* state in the PDA.

The PDA for the above example is



4.5.3 Trie based constructions in `gtb`

We get `gtb` to construct an Aycock and Horspool trie based push down automaton using the method

```
ah_trie[my_grammar]
```

As a side effect of this method `gtb` outputs a VCG rending of the tries in the file

```
trie.vcg
```

Initially the grammar must be terminalised so that there is no recursion other than non-hidden left recursion. As for the RCAs we use the method `terminalise_grammar[]` to produce a terminalised grammar. We can run the RIGLR recogniser on the trie based automata using the method call

```
ri_recognise[this_ah_trie STRING]
```

For example, the script

```
(* ex17 *)
S ::=  ’a’ ~S ’b’ | A ’a’ A | S ’a’ .
A ::=  ’a’ ~B | ’a’ ~B ~B | # .
B ::=  B A | # .

(
ex17_grammar := grammar[S tilde_enabled]
terminalise_grammar[ex17_grammar terminal]
ah_ex17 := ah_trie[ex17_grammar]
ri_recognise[ah_ex17 "aaaabb"]
)
```

constructs the tries

and generates the following output

```
Terminalising B
Terminalising B
Terminalising B
Terminalising S

******: RIGLR recognise: ’aaaabb’
******: RIGLR recognise: accept
Call graph has 24 nodes and 77 edges
```

Chapter 5

Library grammars

Grammars for ANSI-C, ISO-7185 Pascal and a version of IBM VS-COBOL are included with the `gtb` distribution. These are the grammars on which the experiments that have been reported in [JSE04] are based. Basic `gtb` scripts containing these grammars are in the `lib_ex` subdirectory and are called

```
ansi_c.gtb        pascal.gtb     cobol.gtb
```

respectively.

The grammar for ANSI-C has been extracted from [KR88], the grammar for Pascal has been extracted from the ISO Standard and the grammar for COBOL is from the grammar extracted by Steven Klusener and Ralf Laemmel, which is available from `http://www.cs.vu.nl/grammars/vs-cobol-ii/`.

The grammars have been put into a BNF form using our `ebnf2bnf` tool, followed by some manual manipulation.

Also in the `lib_ex` directory are token strings on which the recognisers for these grammars can be run. The strings have been obtained from original programs, written for other purposes, which have been ‘tokenised’ so that an initial lexical analysis phase is not needed.

These token strings are as follows:

- `bool.tok`
  An ANSI-C program implementing a Quine-McCluskey Boolean minimiser. It contains 4,291 tokens.

- `rdp_full.tok`
  An ANSI-C program formed from the source code of our RDP tool. It contains 26,551 tokens.

- `gtb_src.tok`
  An ANSI-C program formed from the source code of the GTB tool itself. It contains 36,827 tokens.

- `treeview.tok`
  A Pascal program designed to allow elementary construction and visualisation of tree structures. It contains 4,425 tokens.

- ◇ `view_ite.tok`
  The `treeview` program with the `if-then` statements replaced by `if-then-else` statements whose `else` clause is empty. This allows a longest match LR(1) parser to successfully parse the string. It contains 4,480 tokens.

- ◇ `quad.tok`
  A short Pascal program that calculates quadratic roots of quadratic polynomials. It contains 279 tokens.

- ◇ `cob_src.tok`
  A Cobol program based on one of the functions available from `http://www.cs.vu.nl/grammars/vs-cobol-ii/`. It contains 2,197 tokens.

The `gtb` parse functions can read the input string from a file. To do this simply open for reading the file that contains the token string and supply that file in place of the STRING argument.

```
this_derivation :=
   rnglr_recognise[this_dfa this_dfa open["bool.tok" read_text]]
```

NOTE: care is needed here because the default option for the `open` method is `write_text` so if the `read_text` option is left out then the file, in this case `bool.tok`, will be overwritten with an empty file!

Also note that `gtb` script files for each of the example grammars discussed in this guide are included in the `tut_ex` subdirectory.

Chapter 6

`gtb` methods

`ah_trie[my_grammar]` p98
Builds an RCA using the Aycock and Horspool trie based method.

`augment_grammar[my_grammar]` p22
Augments the grammar `my_grammar` if it does not already have a start rule of the required form.

`close["file_name"]` p15
Closes the file called `file_name` previously opened for reading and writing.

`cycle_break_sets[my_gdg prune_break_sets_by_table 0]` p88
Finds the strongly connected components and minimal terminalisation sets for the graph `my_gdg`. Replacing the `prune_break_sets_by_table` option with `retain_break_sets` causes all the sets constructed to be listed. Replacing 0 with N causes only sets of size up to N to be constructed.

`dfa[my_nfa]` p28
Constructs a DFA from the NFA `my_nfa`.

`gdg[my_grammar]` p14
Builds a grammar dependency graph for `my_grammar`.

`generate[my_grammar 10 left sentences]` p5
Constructs 10 sentences from the grammar `my_grammar` using left-most derivations. The parameter `left` can be replaced by `right` and `random`. The parameter `sentences` can be replaced by `sentential_forms`.

`grammar[start_symbol tilde_enabled]` p5
Creates a grammar from the given rules taking `start_symbol` as the start symbol. The `tilde_enabled` flag is an optional flag that creates the additional grammar symbols from a tilded rule set.

`gtb_verbose` p30
Switches on and off verbose diagnostics.

`la_merge[my_dfa]` p43
Constructs the LALR DFA from the LR(1) DFA `my_dfa`.

`lr_parse[my_dfa STRING]` p29
Parses the string `STRING` using an LR parser and the DFA `my_dfa`. If STRING is replaced with a file name then the string is read from the file.

```
nfa[my_grammar lr 1 terminal_lookahead_sets
                          full_lookahead_sets normal_reductions]
```
p23
Creates an NFA from the grammar `my_grammar`. The parameter `lr` can be replaced with `unrolled` to get an IRIA NFA. The parameter 1 can be replaced by 0, to get an LR(0), or 0-1, to get an SLR(1) NFA. The last three parameters are optional and the ones given are the defaults. There is one other possibility for each of these parameters, `non-terminal_lookahead_sets`, `singleton_lookahead_sets` and `nullable_reductions`, respectively.

`nfa[my_grammar slr 1]` p35
This is an abbreviation for `nfa[my_grammar lr 0-1 terminal_lookahead_sets full_lookahead_sets normal_reductions]`

`open["file_name" write_text]` p12
Opens a file called `file_name` for writing. To open for read change the option `write_text` to `read_text`. The default option is `write_text`.

`prefix_grammar[my_grammar]` p94
Constructs a left context prefix grammar from `my_grammar`.

`render[my_file my_grammar]` p12
Writes a file based on its second argument to the file `file_name` where this file is created using `my_file := open["file_name"]`

`ri_recognise[my_grammar STRING]` p83
Recognises the string `STRING` using an RIGLR parser and the grammar `my_grammar`. If STRING is replaced with a file name then the string is read from the file.

`rnglr_recognise[my_dfa STRING]` p61
Recognises the string `STRING` using an RNGLR parser and the DFA `my_dfa`. The DFA must be RN. If STRING is replaced with a file name then the string is read from the file.

`terminalise_grammar[my_grammar terminal]` p79
Convert non-terminals written `~A` in the grammar into terminals. To turn them back replace the `terminal` option (which is the default option) with `nonterminal`.

`tomita_1_parse[my_dfa STRING]` p51
Parses the string `STRING` using a Tomita parser and the DFA `my_dfa`. If STRING is replaced with a file name then the string is read from the file.

`write[my_grammar]` p8
Prints output based on its argument to the primary output device.

Bibliography

[AH99] John Aycock and Nigel Horspool. Faster generalised LR parsing. In *Compiler Construction, 8th Intnl. Conf, CC'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 32 – 46. Springer-Verlag, 1999.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles techniques and tools.* Addison-Wesley, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 — Parsing of *Series in Automatic Computation.* Prentice-Hall, 1972.

[DeR69] Franklin L DeRemer. *Practical translators for LR(k) languages.* PhD thesis, Massachussetts Institute of Technology, 1969.

[DeR71] Franklin L DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, July 71.

[GJ90] Dick Grune and Ceriel Jacobs. *Parsing Techniques: A Practical Guide.* Ellis Horwood, Chichester, England. (See also: `http://www.cs.vu.nl/~dick/PTAPG.html`), 1990.

[JSE04] 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, *Proc. 4th Workshop on Language Descriptions, Tools and Applications LDTA2004*, also in Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[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.

[NF91] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 60–75. Kluwer Academic Publishers, Netherlands, 1991.

[Rek92] Jan G. Rekers. *Parser generation for interactive environments.* PhD thesis, Universty of Amsterdam, 1992.

[San95] Georg Sander. *VCG Visualisation of Compiler Graphs.* Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.

[SJ02] Elizabeth Scott and Adrian Johnstone. Table based parsers with reduced stack activity. Technical Report TR-02-08, Computer Science Department, Royal Holloway, University of London, London, 2002.

[SJ05] Elizabeth Scott and Adrian Johnstone. Generalised bottom up parsers with reduced stack activity. *The Computer Journal*, 48(5):565–587, 2005.

[SJ06] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4), 2006.

[SJH00] Elizabeth Scott, Adrian Johnstone, and Shamsa Sadaf Hussain. Tomita-style generalised LR parsers. Updated Version. Technical Report TR-00-12, Computer Science Department, Royal Holloway, University of London, London, December 2000.

[Tar72] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146:160, 1972.

[Tom86] Masaru Tomita. *Efficient parsing for natural language.* Kluwer Academic Publishers, Boston, 1986.

[Tom91] Masaru Tomita. *Generalized LR parsing.* Kluwer Academic Publishers, Netherlands, 1991.