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

1 gtb overview

There are many aspects motivating the investigation of context free grammars, for example to find better general parsing 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 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].

This is a brief guide to using the gtb tool kit, it is essentially a cut down version of the more detailed gtb tutorial guide. It is aimed at users who are familiar with the theory of parsing and with the use of parser generators. Standard definitions of finite state automata, context free grammars, derivations and related concepts will be assumed, as will familiarity with the standard LR parse tables and stack based parsers. For a full guide to gtb and the related underlying theoretical concepts the reader is referred to the much more detailed gtb tutorial guide.

1.1 gtb input files

The input to gtb consists of a set of grammar rules and an LC script of methods to be executed. 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 a small grammar, ex1.

```
(* ex1 *)
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .
(* the LC instructions are enclosed in parantheses *)
(
ex1_grammar := grammar[S]
generate[ex1_grammar 15 right sentential_forms]
)
```

Comments are contained within brackets of the form (* *) and the gtb method calls are contained within parentheses. The method grammar[S] causes gtb to make a grammar using the specified rules and start symbol S, in the above example the grammar is referred to using the variable name ex1_grammar. The second method in the above example gets gtb to generate 15 strings using right-most derivations.

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 also include non-terminals.

The write method gets gtb to print out some of the data structures 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

```
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .
(
gtb_verbose := true
write[grammar[S]]
)
```

diagnostic information is printed on the screen.

```
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] .

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 . }
```

Internally all of the grammar symbols are given unique integer numbers, listed at the top of the output. 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. Finally `gtb` outputs its versions of the FIRST and FOLLOW sets for each symbol.

1.2 Enumeration and the rules tree

`gtb` represents the grammar that it constructs using a rules tree. The internal rules tree can be output to a file in VCG format.

```
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 that is displayed in VCG as

1.3 Grammar dependency graphs

For nonterminals A and B, the relationship A *depends on* B is defined by the property that B appears on the right hand side of the rule for A. The relation A depends on B is represented in a *grammar dependency graph* (GDG). For example,

```
S ::= S '+' S | S '*' S | E .
E ::= 'a' | 'b' .
(
ex1_grammar := grammar[S]
write[ex1_grammar]
render[open["gdg.vcg"] gdg[ex1_grammar]]
close[gdg_file]
)
```

builds the following 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.

2 Using `gtb` – a quick start

You can download `gtb` from

`www.cs.rhul.ac.uk/research/languages/projects/gtb/gtb.html`

The distribution contains sub-directories so the file needs to be unpacked with the appropriate sub-directory flag. The WINDOWS distribution contains executables `gtb_xp` and `gtb_cygwin` that run from the command line under Windows XP and under cygwin. There is also a UNIX distribution and the source files can be compiled to run under other systems, see the README file. Some example files are included with the distribution. Typing `gtb_xp \gtb_ex\ex1` will provide an initial example of how the tool works.

To use `gtb` on your own examples simply create an input file `file.gtb` in which the grammar rules are listed at the top of the file in the form described in Section 1.1 and then the `gtb` methods you want to execute are listed, in order, enclosed in parantheses.

`gtb` can generate a specified number of sentences or sentential forms from the grammar. It can build LR(0), SLR, LALR, LR(1) and Aycock and Horspool style automata and it performs standard LR parses and various forms of GLR parses. It constructs various types of grammar related structures such as FIRST and FOLLOW sets and it can be used to detect recursion and to terminalise a grammar to remove recursion. `gtb` can be instructed to output textual information to the screen and graphical data structures in VCG format.

The following is an example `gtb` input script

```
S ::=  E ';' .
E ::= E '+' T | T .
T ::= '0' | '1' .
X ::=  'a' ~X 'b' | A 'a' A | X 'a' .
A ::=  'a' ~B | 'a' ~B ~B | # .
B ::=  B A | # .
(
this_grammar := grammar[S]
gtb_verbose := true
this_nfa := nfa[this_grammar lr 1]
this_dfa := dfa[this_nfa]
this_lalr := la_merge[this_dfa]
lr_parse[this_lalr "0+1;"]
gtb_verbose := false

this_derivation := tomita_1_parse[dfa[nfa[grammar[S] slr 1]] "0+1;"]
render[open["gss.vcg"] this_derivation]

next_grammar := grammar[X tilde_enabled]
terminalise_grammar[next_grammar terminal]
ah_ex := ah_trie[next_grammar]
ri_recognise[ah_ex "aaaabb"]
)
```

The methods construct an LR(1) DFA for the grammar whose start non-terminal is S, then construct an LALR table by merging appropriate states, and then run the standard LR parse algorithm on the input string $0 + 1;$. The directive `gtb_verbose:=true` causes diagnostics on the behaviour of these methods to be printed on the screen. The method `tomita_1_parse` performs a Tomita-style parse of the same string using the SLR(1) parse table and the `render` method causes the graph structured stack constructed during this parse to be written to a VCG input file. The method `terminalise_grammar` causes the grammar whose start symbol is X to be terminalised according to the user-supplied ~ annotations, and the final two methods build the required automaton and run a reduction-incorporated parse using this automaton on the string *aaaabb*.

We complete this section with a full listing of the current `gtb` methods. These methods are discussed briefly in the following sections and full details can be found in the comprehensive Tutorial Guide.

2.1 The `gtb` methods

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

`augment_grammar[my_grammar]`

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

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

`cycle_break_sets[my_gdg prune_break_sets_by_table 0]`
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]`
Constructs a DFA from the NFA `my_nfa`.

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

`generate[my_grammar 10 left sentences]`
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]`
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`
Switches on and off verbose diagnostics.

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

`lr_parse[my_dfa STRING]`
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]
```
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]`
This is an abbreviation for `nfa[my_grammar lr 0-1 terminal_lookahead_sets full_lookahead_sets normal_reductions]`

`open["file_name" write_text]`
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]`
Constructs a left context prefix grammar from `my_grammar`.

`render[my_file my_grammar]`
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]`
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]`
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]`
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]`
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]`
Prints output based on its argument to the primary output device.

3 LR automata

The standard LR(0), SLR(1) and LR(1) parsing automata can be built by first constructing a nondeterministic finite automaton (NFA) from the grammar rules and then applying the subset construction to get a DFA, see for example [GJ90] or the `gtb` tutorial guide.

For LR automaton construction the grammar needs to be augmented, so that the rule for the start symbol is of the form $S ::= A$. A `gtb` function that requires an augmented grammar will test its input and augment it if necessary. It is possible to explicitly instruct `gtb` to augment a grammar using the method `augment_grammar[my_grammar]` which modifies the grammar rather than creating a new, independent grammar.

3.1 LR NFA construction

The script

```
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.

To generate SLR(1) and LR(1) DFAs replace `lr 0` with `slr 1` and `lr 1` respectively. (There are some technical issues relating to the construction of an LR(1) NFA. These are discussed in the tutorial guide in the section on the singleton set NFA model.)

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).

3.2 LR DFA construction

For pedagogic purposes, `gtb` constructs the LR(0) DFA from the NFA as using the subset construction.

```
S ::=  A 'b' | 'a' 'd' .
A ::=  A 'a' | # .
(
ex3_grammar := grammar[S]
ex3_dfa := dfa[nfa[ex3_grammar lr 0]]
render[open["dfa.vcg"] ex3_dfa]
)
```

generates the following VCG output DFA

3.3 Deterministic LR parsing

`gtb` has an standard LR parser which uses the LR DFAs. The parser is run using a specified DFA and a specified input string using the method `lr_parse[my_dfa STRING]`

```
S ::=  E ';' .
E ::= E '+' T | T .
T ::= '0' | '1' .
(
ex2_grammar := grammar[S]
gtb_verbose := true
lr_parse[dfa[nfa[ex2_grammar lr 0]] "0+1;"]
gtb_verbose := false
)
```

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.

```
******: 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. 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.

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` using the method `write[open["parse.tbl"] my_dfa]`

3.4 LALR DFAs

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`.

```
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]
)
```

4 GLR algorithms

The problem with the standard stack based LR parsers is that the LR parse tables for many grammars contain conflicts. One possible approach to dealing with conflicts is to pursue each of the choices in parallel, generating a separate stack for each process. Tomita [Tom91] devised a method for combining the multiple stacks so that they require at most quadratic space. This allowed him to give a practical, generalised version of the LR parsing algorithm. 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. Tomita's original 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.

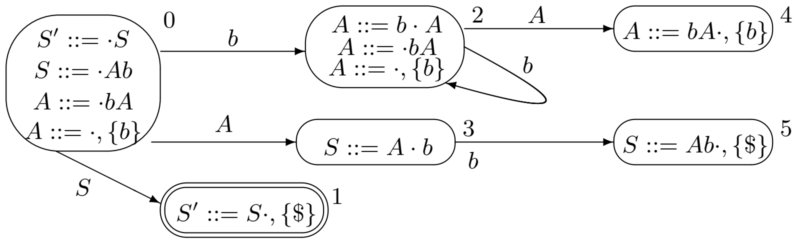
4.1 Building a GSS

Tomita's original exposition pushes the grammar symbols onto the stack between the state symbols. 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.

A discussion of Tomita's algorithm can be found in the `gtb` tutorial manual or, for example, in [Tom91], [SJH00] or [SJ06]. Here we just illustrate the approach with an example.

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

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. 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
```

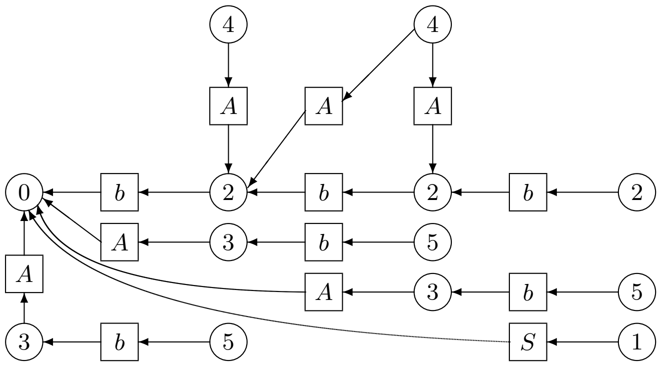
Next we apply the shift action to states 2 and 3. The other two stacks die. Applying the reduction to the stack $0b2b2$ gives the stack $0b2b2A4$, then applying the reduction in state 4 (twice) results in the stacks

```
               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
```

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. In the above example 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.)



4.2 Tomita's algorithm in `gtb`

We can run Tomita's algorithm on an LR DFA using the method `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 an LR(0), SLR(1), LALR or LR(1) DFA.

```
S ::=  A 'b' .
A ::=  'b' A | # .
(
this_derivation := tomita_1_parse[dfa[nfa[grammar[S] lr 1]] "bbb"]
render[open["gss_ex8.vcg"] this_derivation]
)
```

As a result of running the Tomita parser `gtb` produces a stack structured graph that is a version of the GSS. 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 parse using `gtb_verbose`.

4.3 Right Nulled parse tables

Tomita's original algorithm does not always correctly parse an input string if the grammar contains right nullable rules, that is rules of the form $A ::= \alpha\beta$ where $\beta \overset{+}{\Rightarrow} \epsilon$. 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 an RN 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 include the right nullable reductions in a `gtb` generated DFA we use the method call `nfa[my_grammar lr 1 nullable_reductions]`. We then run the subset construction using the `dfa` method exactly as for the standard LR case.

For example, running the script

```
S ::=  'b' A .
A ::=  'a' A B | # .
B ::=   # .
(
rn_nfa := nfa[grammar[S] lr 1 nullable_reductions]
render[open["nfa1.vcg"] rn_nfa]
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, NFA and DFA

The following diagnostics are also produced, showing that the first input string is correctly accepted and that the other three 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. This allows the performance of Tomita's algorithm to be compared with other algorithms. To get more detailed diagnostics as the parse proceeds we can switch on the verbose mode.

4.4 The RNGLR algorithm

The RNGLR algorithm [SJ06] is essentially similar to Tomita's GLR algorithm but it is more efficient in the case of right nullable rules. Also, the GSS constructed by the RNGLR algorithm does not have symbol nodes 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. To run the RNGLR algorithm in `gtb` we use the method `rnglr_recognise[my_dfa STRING]`.

4.5 The RNGLR parser

A parser is a recogniser that outputs, in some form, a derivation of the input string, if that string is in the language. 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. 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.

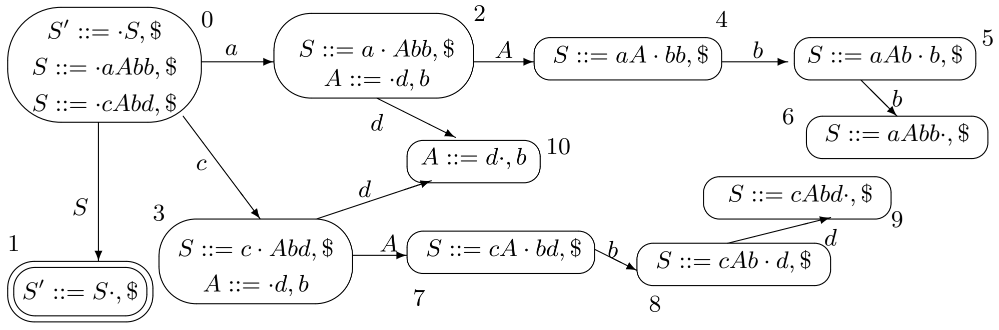
To run the parser version of the RNGLR algorithm in `gtb` we use the method `rnglr_parse[my_dfa STRING]`.

5 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 forms the basis of a different type of general parsing algorithm [AH99].

Consider the grammar, ex11,

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



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. However, we could add a reduction transition (a special form of ϵ-transition that does not require reading an input symbol) from state 6 to state 1 and traverse this as though it were an ϵ-transition. This would mean that we did not have to push and pop the intermediate states. It turns out that such transitions can always be added except in cases of self embedding.

To deal with different occurances of a nonterminal in different alternates of rules we 'multiply out' the DFA states. A brief description is given in the next section, full details can be found in in [SJ02] or [SJ05].

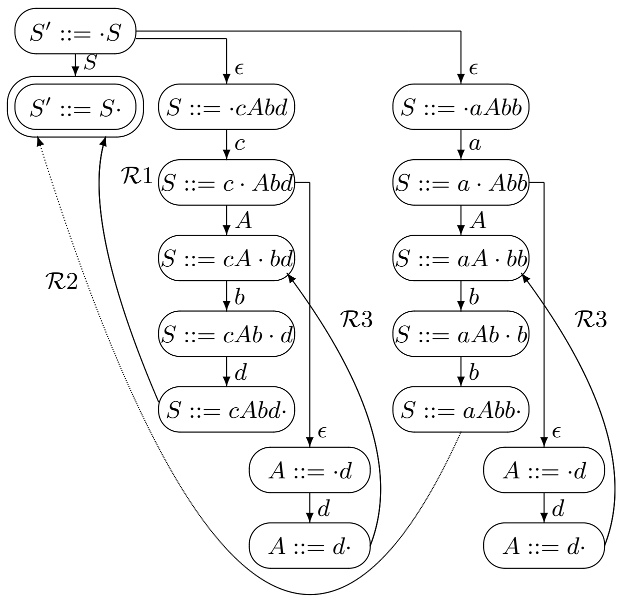
5.1 Reduction incorporated automata

For a grammar that does not contain recursion we construct an automaton, IRIA(Γ), as follows.

Construct 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, labelled $x ::= \cdot\gamma$ and an ϵ-transition from h to t.

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.



The node labelled $A ::= \cdot d$ has been multiplied out to allow for the two instances of the non-terminal A in the grammar rules.

In the case of recursive rules we cannot simply use the multiplying out (unrolling) 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. To instruct `gtb` to build the IRIA for a grammar we use the method `nfa[my_grammar unrolled 0]`

```
S ::=  S 'a' | A .
A ::=  'b' A | # .
(
ex12_iria:= nfa[grammar[S] unrolled 0]
render[open["nfa.vcg"] ex12_iria]
)
```

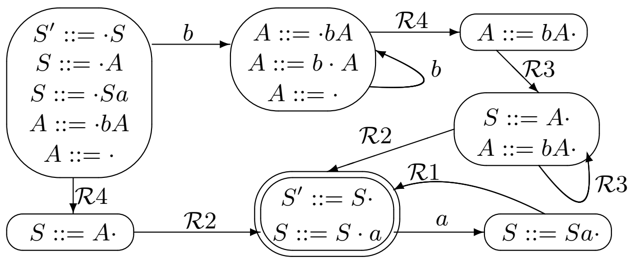
creates the IRIA

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.

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.

5.2 Reducing non-determinism – RIA(Γ)

The RIGLR algorithm is more efficient if there is less non-determinism in the IRIA automaton. 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}).



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

5.3 Terminalising a grammar

We now build a push down automaton, RCA(Γ), for any given context-free grammar, Γ. We begin by modifying the grammar to remove most of the recursion by creating 'terminalised' instances, $A^{\perp}$, of recursive non-terminals creating a new grammar Γ_S (this changes the language generated by the grammar). We then construct the RIA for Γ_S and for each terminalised non-terminal, and then call these automata from the main RIA. 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. 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.

We can terminalise a grammar, replacing `~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. Running the script

```
S ::=  A 'a' .
A ::=  'a' B | 'a' .
B ::= ~A 'c' .
```

```
(
ex13_grammar := grammar[S tilde_enabled]
terminalise_grammar[ex13_grammar terminal]
iria_S := nfa[ex13_grammar unrolled 0]
render[open["nfa.vcg"] iria_S]
)
```

builds the IRIA

Since the grammar as been mutated the original unterminalised grammar is lost. To reconstruct it use the `nonterminal` option with the `terminalise_grammar` function.

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

We shall call a grammar that has, possibly, terminalised versions of some of its non-terminals a *terminalised grammar*.

5.4 RCA(Γ)

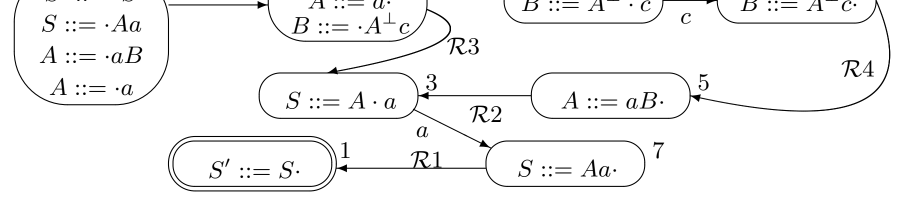
Given a grammar, Γ, terminalise instances of non-terminals so that the resulting grammar, Γ_S say, has no self embedding. For each non-terminal, $A \neq S$, such that Γ_S contains at least one terminalised instance of A, construct the grammar Γ_A that has the same rules as Γ_S but start symbol A.

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.

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). 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. 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.

The following PDA, RCA(Γ_{13}), constructed from the derived grammar, Γ_S,

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



5.5 The RIGLR algorithm

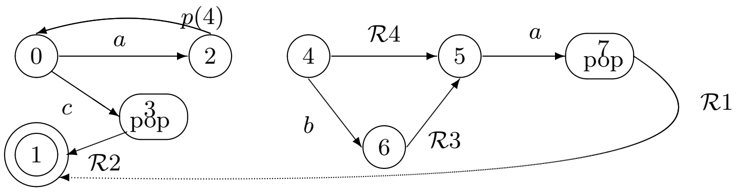
The RIGLR algorithm takes as input an RCA for a grammar Γ and a string $u = a_1 \dots a_d$ and it traverses the RCA using u. It starts in the RCA start state and at each step i it constructs the set U of all RCA states that can be reached on reading the input $a_1 \ldots a_i$. When a push transition 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 $\textsc{follow}(A)$ 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 $\textsc{follow}(A)$.

For example, consider the grammar ex14

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

which has RCA

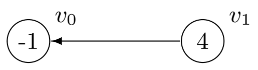


and consider the input string $aacbaa$.

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

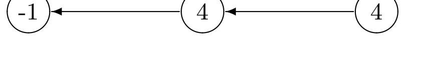
$$U = \{(0, v_0)\}$$

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.



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

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.



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

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 $\textsc{follow}(B)$.

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

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.

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

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]
```

which takes a grammar with terminalisation annotation, generates the structures needed for an RIGLR recogniser and then runs it on the specified STRING.

6 Terminalising a grammar automatically

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 a *terminalisation* of the grammar, which is *minimal* if no proper subset of it is also a terminalisation.

The GDG constructed by `gtb` has been designed to facilitate the identification minimal terminalisations. A 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. 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.

6.1 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 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.

In order to remove recursion, we 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.

Any LR-path from a node to itself in a GDG is contained in a maximal SCC. Each SCC that contains at least one edge labelled L and at least one edge labelled R must be 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.

We begin by using Tarjan's algorithm [Tar72] to find the SCCs. 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. We then consider each LR-SCC in turn. We find all the cycles, and then we consider all the L-cycles. 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. We then remove the non-minimal sets by removing any set which has one of the other sets as a proper subset.

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.

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

6.2 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 on the initial GDG and, as a side effect, `gtb` creates a new version of the GDG in which the nodes in each SCC and the edges between them are identified, the nodes in SCCs of greater than one element are all coloured the same colour, and the edges between nodes in the same SCC 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,

```
S ::= A .
A ::= E E B C .
B ::= 'a' A E 'a' | # .
C ::= 'a' D .
D ::= C D 'a' | # .
E ::= A S | # .
(
ex16_gdg := gdg[grammar[S]]
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
```

Within the VCG tool there is a 'Folding' option which allows nodes and edges to be combined and hidden. 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.

```
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.

6.3 Pruning the search space

In some cases the number of interim terminalisation sets constructed 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]
```

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 a 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]`.

7 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.

7.1 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 set of these strings 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.

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, 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 terminal string w.

Then the language of the LR(0) DFA is the set of all strings

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

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.) For example, running the script

```
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 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] .
```

7.2 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 $(\alpha, \beta, A) \in \Phi(\Gamma)$.)

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

$$[S'] ::= \epsilon \qquad [S] ::= [S'] \mid [S] \qquad [A] ::= [S] \mid [S]\ A\ a$$

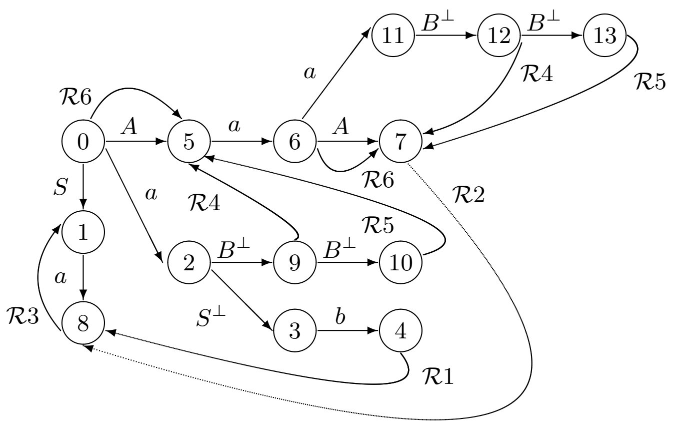
the languages are

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

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

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

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



To build the full PDA, 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. Construct a prefix grammar and a trie based automaton for A, as described above for S'. Then, 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, and remove the nonterminal transitions.

7.3 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

```
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

8 Library grammars

Grammars for ANSI-C, ISO-7185 Pascal and a version of IBM VS-COBOL are included with the `gtb` distribution in the `lib_ex` subdirectory.

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.

`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.

```
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 will be overwritten with an empty file!

References

[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.

[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.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[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.

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