Tomita-Style Generalised LR Parsers

Elizabeth Scott, Adrian Johnstone, and Shamsa Sadaf Hussain

December 24, 2000

Royal Holloway
University of London

Department of Computer Science
Egham, Surrey TW20 0EX, England

Abstract

Roughly speaking a parser is a function, associated with a grammar, which takes as input a string of symbols and produces as output the derivations of that string if it is in the language generated by the grammar and an error message otherwise. In this report we undertake a theoretical study of a general parsing technique based on the standard linear LR parsing algorithm. This general technique was described and given a practical implementation by Tomita and a corrected version was given by Farshi. We shall show that the problem with Tomita's original algorithm lies mainly (but not entirely exclusively) with grammars that contain hidden *right* recursion. The issue is to ensure that when a reduction using a right nullable rule is applied, all the possible reductions are explored. We address this by treating items of the form $A ::= \alpha \cdot \beta$, where $\beta \overset{*}{\Rightarrow} \epsilon$, as reductions, allowing the reduction to be performed when only α has be recognised. We give a modification of Tomita's original algorithm, based on a modification of the underlying parse table, which we prove is correct. It is also more efficient than Farshi's modification. We have implemented a parser generator which generates parsers based both on Tomita's original algorithm and on our modification, and we give statistics on the behaviours of both types of parser.

Contents

1 Background **2**
1.1 Traversing a DFA using a given input string 3

2 The graph structured stack (GSS) **5**
2.1 Notation and properties of the GSS 6
2.2 Constructing a GSS for input $a_1 \dots a_n$ 7
2.3 Dealing with ϵ-rules 11
2.4 Further issues to be considered 12

3 Tomita's algorithms **13**
3.1 Processing reductions 13
3.2 Storing pending reductions 16
3.3 Tomita's first stage algorithm 17

4 Tomita's algorithm applied to general grammars **19**
4.1 Algorithm 1e 19
4.2 An example with hidden left recursion 20
4.3 The (in)correctness of Algorithm 1e 23
4.4 Hidden right recursion 24
4.5 Right nullable rules 26
4.6 Merging symbol nodes 31

5 Farshi's algorithm **34**
5.1 Farshi's recognition algorithm for general grammars 34
5.2 The efficiency of Farshi's algorithm 35
5.3 A modified version of Farshi's algorithm 37

6 GLR parsing with a modified DFA **39**
6.1 The reduction modified DFA and an example 39
6.2 Generalised reduction modified LR parser (GRMLR) 41
6.3 The efficiency of the algorithm 43
6.4 Correctness of the algorithm 44
6.4.1 RM-Parse tables 45
6.4.2 Table based parsers 45
6.4.3 Equivalence of LR(1)- and RM-parsers 46
6.4.4 Correctness of the GRMLR algorithm 49
6.5 The reduction modified DFA and LR(1) grammars 55
6.5.1 The correctness of reduced RM-tables for LR(1) grammars 56
6.5.2 Resolved RM-parse tables 58

7 Experimental results **62**
7.1 The experiments 62
7.2 The results 63

Tomita-Style Generalised LR Parsers

Roughly speaking a parser is a function, associated with a grammar, which takes as input a string of symbols and produces as output the derivations of that string if it is in the language generated by the grammar and an error message otherwise. It is sometimes claimed that parsing is 'done'. Such claims are somewhat surprising, particularly when we recall that it is still not known whether all context free grammars, or even all context free languages, admit a linear time parser. It is certainly the case that parsing is well enough understood for efficient parsers to be easily generated in many cases, but being able to build a bridge which we are fairly sure will not fall down is not the same thing has having a fully understood mathematical theory of forces. For example, the grammars normally used for C do not fall into the class of grammars for which there is a standard linear time parser. In this case the problem is with the ambiguity of certain constructs such as `if_then_else`. There are practical solutions to these problems, in the case of the `if_then_else` ambiguity one of the many different possible derivations is selected (the one which corresponds to the 'longest match'). Such solutions provide parsers which behave in a way which is satisfactory for the end users, but the 'parsers' do not produce all derivations for a given input string and hence are theoretically incomplete. Furthermore, it is relatively easy to construct a grammar which does not have the properties required for the known linear-parser generation techniques to be applied (i.e. a non-LR grammar), and it is often difficult to transform such a grammar into an appropriate form.

A properly understood underlying mathematical model which allows reasoning about the behaviour of objects and which feeds into good engineering practice is the basis of any scientific academic subject. The above observations highlight two areas where the underlying theory of parser generation needs to be extended. We need a better understanding of the properties which make a grammar LR(1), in order to allow such grammars to be effectively constructed by non-specialists, and we need further study of techniques for constructing parsers for non-LR(1) grammars. The latter is particularly important in cases where we are not free to change the language to make it more amenable to parsing, for example in natural language parsing.

In this report we undertake a theoretical study of a general parsing technique based on the standard linear LR parsing algorithm. This general technique was described and given a practical implementation by Tomita [Tom86]. However, there was no mathematical analysis of the algorithm and it was subsequently found to be non-terminating in certain cases. Farshi [NF91] produced a modification of Tomita's algorithm which appears to correct the problem, but Farshi himself states: "Although no formal proof was provided here but it is believed that the modified algorithm is a precompiled equivalent of Earley's algorithm with respect to its coverage."

There is another algorithm given by Nederhof and Sarbo [NS96] which addresses the problem with Tomita's algorithm by essentially carrying out ϵ removal 'on the fly'. The paper includes a proof of the correctness of the algorithm. However, the focus of the paper is on the problem of hidden left recursion, which Nederhof and Sarbo claim is also a problem for Nederhof's cancellation parsing algorithm. Since we shall show that the problem with Tomita's algorithm can be successfully addressed by considering right nullable production rules, we shall not discuss Nederhof and Sarbo's algorithm further in this report.

Tomita constructed his algorithm in stages. The first stage algorithm was applied only to ϵ-free grammars. This was then extended to general context-free grammars, but the extension was later found to fail to terminate on grammars with hidden left recursion. We shall study Tomita's first stage algorithm and explain the problem with applying it to grammars which contain ϵ-rules. As a result of this analysis we shall give a modification, based on a modification of the underlying parse table, which we can prove is correct. It is also more efficient than Farshi's modification. We shall show that the problem with Tomita's first stage algorithm lies mainly (but not entirely exclusively) with grammars that contain hidden *right* recursion. The issue is to ensure that when a reduction using a right nullable rule is applied, all the possible reductions are explored. We address this by treating items of the form $A ::= \alpha \cdot \beta$, where $\beta \overset{*}{\Rightarrow} \epsilon$, as reductions, allowing the reduction to be performed when only α has been recognised.

We have implemented a parser generator which generates parsers based both on Tomita's first stage algorithm and on our modification, and at the end of this report we shall give statistics on the behaviours of both types of parser.

In a later paper we shall discuss a different approach to constructing generalised parsers in which the regular parts of the language are parsed using an efficient finite state automaton, and the self-embedding is handled using recursive calls to sub-automata. Again we shall prove that the parsers constructed using this algorithm are correct.

1 Background

LR parsers use a grammar-related deterministic finite state automaton (DFA) and a stack to parse an input string. The DFA is traversed as input symbols are recognised, and the path taken during this traversal is recorded on the stack. When an accepting state in the DFA is reached, the path must be re-traced up to a certain point and a new route taken (this is performing a 'reduction'). This is implemented by popping the appropriate number of states off the stack.

It is possible for the DFA to present more than one choice of action to the parser, an accepting state may also have a transition on the next input symbol (a shift/reduce conflict), and there may be more than one path re-tracing which is possible (a reduce/reduce conflict). In such cases the standard LR parsing algorithm is inadequate; it does not specify how to make a choice of action when several possibilities exist.

Lang [Lan74] proposed a method for exploring all of the possible actions on a given input string, and outputting a grammar which generates all of the possible derivations of that string. In this report we describe implementations, based on this approach, which have been given by Tomita [Tom86] and Farshi [NF91], and we give another implementation which is essentially the same as Tomita's but which is based on a modification of the underlying DFA, obtained by extending the set of states which are accepting states. In this section we shall discuss the underlying LR(1) parsing technique in a way which will allow us to reason about Tomita's algorithm.

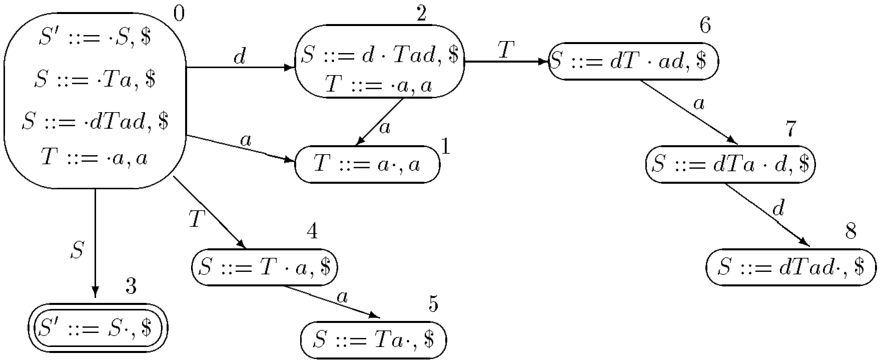
1.1 Traversing a DFA using a given input string

In the following discussion we shall use LR(1) DFAs, but the discussion applies equally well to LR(0), SLR(1) and LALR(1) DFAs.

The basic idea is to maintain a list of DFA states which can be reached by starting at the start state and traversing the DFA using the input symbols seen so far. When an input symbol, a say, is read any state which can be reached from a state in the current list along a transition labelled a is added to the new list of current states. When an accepting state (a state which contains an item $(A ::= \alpha\cdot, b)$ say where b is the next input symbol) is reached the path labelled α which was taken to reach that state is re-traced and the state which can then be reached along a transition labelled A is also added to the list of current states. This process is very similar to the 'subset construction' which is used to construct a DFA from an NFA: at any point in the procedure we have a current set X of states, we make a new set Xa consisting of those states which can be reached from a state in X along a transition labelled a, and then we form the ϵ-closure of Xa, i.e. we construct the smallest set Y with the property that $Xa \subseteq Y$ and if a state n can be reached from a state m in Y along an ϵ-transition then n is in Y.

The main difference between the DFA-based general parser and the NFA traverser which forms the basis of the subset construction is that in the latter the ϵ-closure contains all the states which can be reached via ϵ-transitions, while the reduction-closure in the former can only contain states which can be reached by re-tracing a path taken (we refer to these as *input related reduction-closures*). The following example illustrates this point.

$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & Ta \mid dTad \\ T & ::= & a \end{array}$$



If we construct the reduction-closures which arise from input aad we get:

start position: $\{0\}$, reduction-closure $= \{0\} = S_0$.
input a: (states reachable from S_0) $= \{1\}$, reduction-closure $= \{1, 4, 6\} = S_1$.
input a: (states reachable from S_1) $= \{5, 7\}$, reduction-closure $= \{3, 5, 7\} = S_2$.
input d: (states reachable from S_2) $= \{8\}$, reduction-closure $= \{3, 8\} = S_3$.

The traverser reaches the final accept state, state 3, and has consumed all the input, thus it will erroneously accept the input *aad*. The problem is that, in the second step of the traversal, it is possible to get to state 6 from state 1 via a valid reduction but it is not possible to get there by re-tracing a path which has been taken in the traversal at this stage. Thus we need to restrict reductions and use only the input related reduction closures.

Because we can only perform reductions down paths which have been traversed with the current input it is necessary to retain these paths. This is the role of the stack in a traditional LR parser. In this report we shall consider variations on a method devised by Tomita for recording these paths and computing the states which can be generated from reductions. In a future paper we shall consider an alternative approach, based on the work of Aycock and Horspool [AH99], in which the DFA is modified so that all reductions from a given state and next input symbol are valid. It is then no longer necessary to retain the information about what paths were taken to a given state. In fact the reductions can be pre-calculated and included in the DFA when it is initially constructed.

Tomita's method for recording a possibly multipath traversal of a DFA is based on a *Graph Structured Stack*, which replaces the stack in a traditional LR parser and allows the appropriate input related reduction-closures to be efficiently calculated. The method by which Tomita chose to identify the reductions that needed to be carried out meant that, if the grammar contained right nullable rules, his basic algorithm did not always correctly identify all reductions that needed to be performed. Thus he applied this algorithm only to ϵ-free grammars. Tomita modified his method to allow grammars which contain ϵ, but this modification results in an algorithm which fails to terminate on grammars which contain hidden left recursion. Farshi corrected this problem by returning to Tomita's basic algorithm and augmenting it with a search which ensured that all reductions were identified even in the case of hidden right recursion. However, the GSS and shared derivation tree produced by Tomita's method are more efficient, in general, than those produced by Farshi's method. Rekers [Rek92] uses Farshi's algorithm for generating and traversing the GSS but constructs more efficient shared parse trees. In the next section we shall define and discuss the GSS which forms the basis of all the algorithms discussed in this report. In Sections 3 and 4 we shall discuss Tomita's algorithms and in Section 5 we shall discuss Farshi's modification.

2 The graph structured stack (GSS)

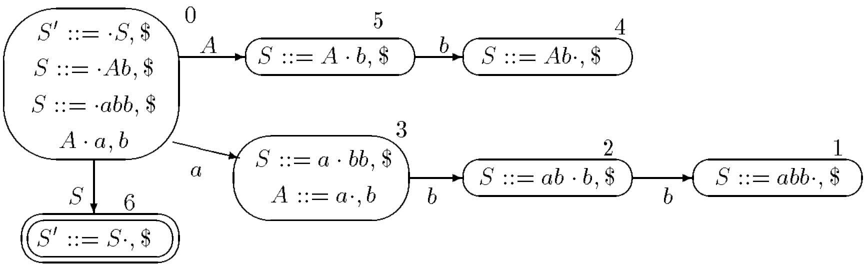
Tomita's algorithm takes an input string, $a_1a_2 \ldots a_n$, and uses it to traverse a DFA, constructing a *Graph Structured Stack (GSS)* as it proceeds. This construction is carried out in a series of steps, an initial step and then one step for each symbol in the input string.

A GSS consists of state nodes, which are labelled with states of the DFA, and a set of symbol nodes, which are labelled with a grammar symbol. The GSS associated with a specific context free grammar and input string $a_1 \ldots a_n$ say. The state nodes are grouped together into disjoint sets, an initial set, U_0, and one set, U_i, for each element a_i of the input string. In the language used in the introduction, U_0 is the input related reduction-closure of the start state of the DFA, and for $1 \leq i \leq n$ we have that U_i is the input related reduction-closure of $U_{i-1}a_i$, the set of all states which can be reached from a state in U_{i-1} along a transition labelled a_i.

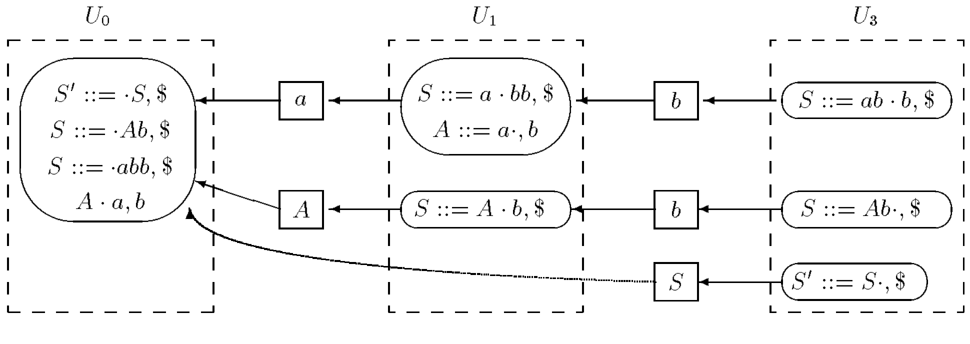
We say that a node is at *level* i if it is in U_i, and that $v \in U_i$ *has a valid reduction* if the DFA state, h, which labels v contains an item of the form $(A ::= \alpha\cdot, a_{i+1})$. Conversely, we say that a reduction via the rule $A ::= \alpha$ is *valid for a state node* v which is at level i and has label h, if the DFA state h contains the item $(A ::= \alpha\cdot, a_{i+1})$. In other words, valid reductions are reductions which can be applied when the input $a_1 \ldots a_i$ has been read and the lookahead input symbol is a_{i+1}.

For example, given the grammar and DFA,

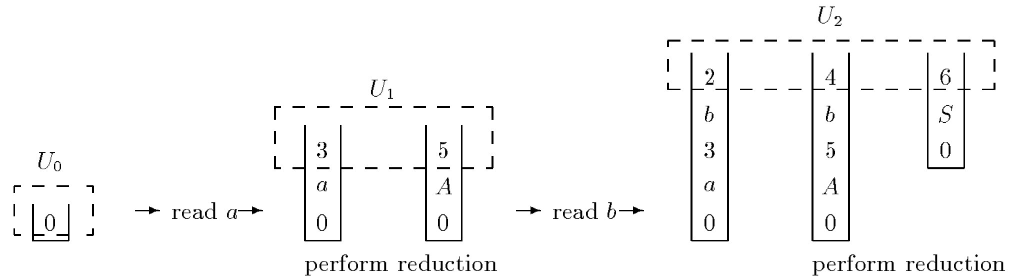
$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & Ab \mid abb \\ A & ::= & a \end{array}$$



the input $ab\$$ results in the GSS



corresponding to the stack activity



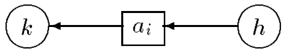
This is obtained by beginning in state 0 with a stack which just contains 0. The input symbol a is read, and shifted on to the stack along with the next state, 3. At this point there are two possible actions, a shift to state 2 and a reduction. The stack is split and both actions are applied. The reduction is applied first, creating a new stack by popping off the top two symbols and pushing on A followed by state 5. In the GSS the base state, 0, of the two stacks is shared. Now the symbol b is read and shifted on to the top of both stacks. There is no further action that can be performed from state 2, but from state 4 there is a valid reduction which results in a third stack splitting. The state 6 is the accepting state of the DFA, so the input is accepted.

We now give some general properties of a GSS, and then we shall describe the general GSS construction, without describing the 'book-keeping' (the identification and storage of pending reductions) that a full algorithm needs to do in order to be able to carry out the construction. The book-keeping will be discussed when we consider the various parsing algorithms.

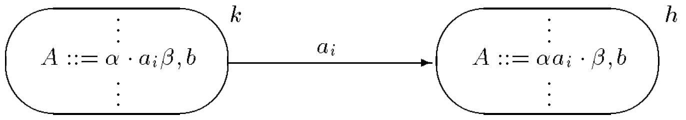
2.1 Notation and properties of the GSS

The GSS is a connected directed bipartite graph on the set of state nodes and the set of symbol nodes, i.e. all successors and predecessors of a symbol node are state nodes and all successors and predecessors of a state node are symbol nodes.

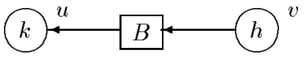
The GSS constructed from input $a_1 \ldots a_n$ contains a subgraph of the form



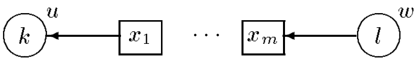
if there is a node in U_{i-1} labelled k, and node in U_i labelled h, and a transition labelled a_i from k to h in the DFA.


$A ::= \alpha \cdot a_i\beta, b$
k
a_i
$A ::= \alpha a_i \cdot \beta, b$
h

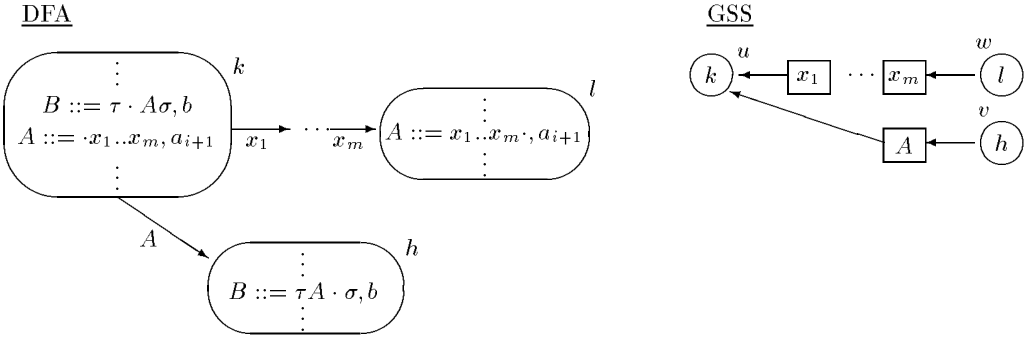
(This corresponds to shifting the symbol a_i in the parser.) The GSS contains a subgraph of the form



where $u \in U_j$ and $v \in U_i$, if and only if there is a subgraph in the GSS of the form



where $u \in U_j$, $w \in U_i$, $A ::= x_1 \ldots x_m$ is a grammar rule, and there is a transition from k to h labelled A in the DFA. (This corresponds to forming an input related reduction from state l.)

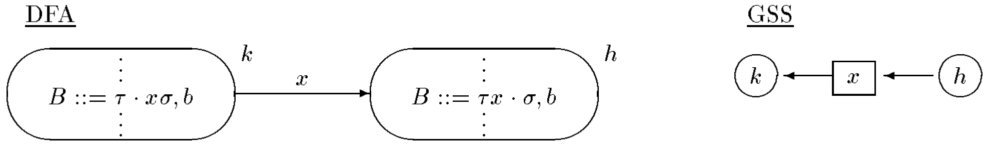

DFA
$B ::= \tau \cdot A\sigma, b$
$A ::= \cdot x_1..x_m, a_{i+1}$
k
x_1
· · ·
x_m
$A ::= x_1..x_m\cdot, a_{i+1}$
l
A
$B ::= \tau A \cdot \sigma, b$
h
GSS
k
u
x_1
· · ·
x_m
w
l
v
A
h

We say that the node v is *reduction related* to the node w via a path of length $2m$ and a symbol node labelled A.

A set of state nodes, U, in the GSS is *input related reduction-closed* if for each node, w say, with label l which is at level i and for each valid reduction in w, i.e. for each item $(A ::= x_1 \ldots x_m \cdot, a_{i+1})$ in l, if k is the label of a state node which is reachable in the GSS from w along a path of length $2m$ then there is a node in U which is at level i and has label h, where h is the state in the DFA which can be reached from k along the transition labelled A. In other words, if for all valid reductions $A ::= x_1 \ldots x_m$ in a node in $w \in U$ there is a node v in U which is reduction related to w via a path of length $2m$ and a symbol node labelled A. The sets U_i in the GSS are all input related reduction-closed.

When the GSS has been constructed the final set, U_n, of state nodes is examined. The input string is in the language if and only if U_n contains the accepting state of the DFA.

Note: Because there is only a path of length 2 in the GSS from a state node with label h to a state node with label k if there is a transition from k to h in the DFA,

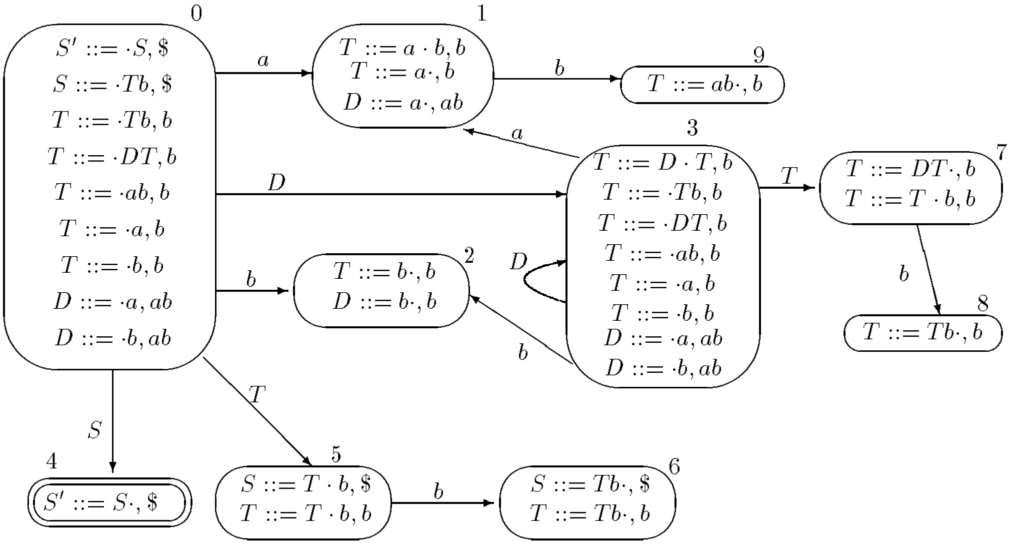


all the symbol nodes which are successors of a given state node in the GSS will have the same label. This label will be the (unique) symbol which labels transitions to h in the DFA.

2.2 Constructing a GSS for input $a_1 \ldots a_n$

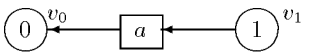
We shall now describe the basic construction of a GSS for a given DFA and input string. (Tomita's basic algorithm does not deal with ϵ-productions and for the moment we shall assume that our grammars are ϵ-free. We shall consider general grammars later.) We illustrate the process using the following example:

$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & Tb \\ T & ::= & Tb \mid DT \mid ab \mid a \mid b \\ D & ::= & a \mid b \end{array}$$

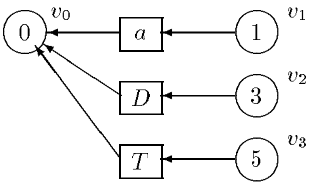


input: *abb*

We begin in the start state, state 0, of the DFA and construct a state node, v_0, labelled 0 in the GSS. If the grammar is ϵ-free then there will be no reductions in state 0 thus $U_0 = \{v_0\}$ and we read the first input symbol, a_1 say. In the DFA we move to state m, say, which is the target of the transition labelled a_1 from state 0. We construct a state node, v_1, in the GSS labelled m, which is added to the set U_1, we construct a symbol node labelled a_1, and we make this symbol node a successor of v_1 and a predecessor of v_0. In our example the input consumed is a and we move to state 1.

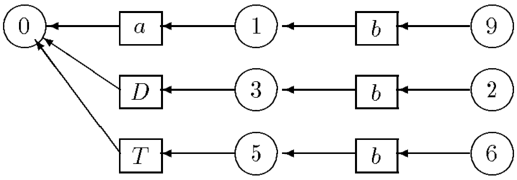


Next we look at state m in the DFA to see if it has any reductions on the next input symbol a_2, i.e. to see if v_1 has any valid reductions. Since we have only consumed one input symbol, such items must actually be of the form $(A ::= a_1\cdot, a_2)$. In our example, the state node labelled 1 contains two reductions $(T ::= a\cdot, b)$ and $(D ::= a\cdot, b)$. We retrace the path of length 2 from v_1 to reach the state node v_0. We then traverse the DFA from state 0 along the transitions labelled D and T to reach states 3 and 5. We create state nodes v_2 and v_3 labelled 3 and 5 and add them to the set U_1, and we create symbols node which has v_0 as a successor and v_2 and v_3 as predecessors.

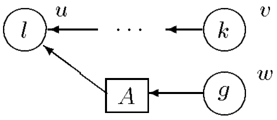


We then check the newly added state nodes for valid reductions, and continue the process until U_1 is input related reduction-closed. In our example, the newly added nodes don't have any valid reductions so this step of the process is complete.

In general, at the beginning of step i we have a set U_{i-1} of state nodes in the GSS, and remaining input $a_i \dots a_m\$$. For each state node, labelled k say, in U_{i-1} we see if there is a transition in the DFA labelled a_i from k to a state m. For each such transition we first check to see if there is a state node in U_i labelled m. If there is not we create such a state node in the GSS and add it to U_i. If the state labelled m has a successor labelled a_i then add an arrow from this node to the node in U_{i-1} labelled k. Otherwise, create a successor node of m labelled a_i and make this node a predecessor of the node labelled m. When all the nodes in U_{i-1} have been dealt with we have $U_i = U_{i-1}a_i$. In our example at step 2 we have $U_1 = \{1, 3, 5\}$ and $U_1 b = \{2, 6, 9\}$.

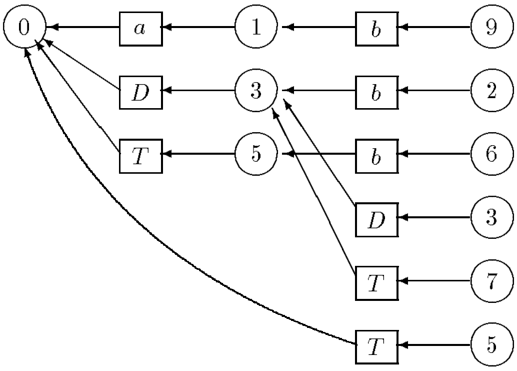


We then form the input related reduction-closure of the set $U_{i-1}a_i$. For each state node, $v \in U_i$, labelled k say, in U_i, and for each item $(A ::= x_1 \dots x_m \cdot, a_{i+1})$ in k, find all the state nodes u, labelled l say, in the GSS which are on a path of length $2m$ from v. Let g be the node reached from l along the transition labelled A in the DFA. If there is not already a state node in U_i labelled g then create such a node, w, and add it to U_i.



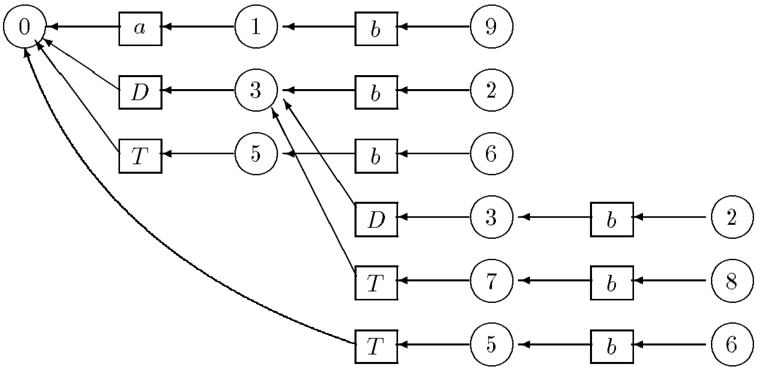
If there is already a path of length 2 from w to u then do nothing. Otherwise create a symbol node labelled A which is a successor of w and a predecessor of u. We continue this process until all the reductions in all the nodes in U_i have been dealt with, i.e. until U_i is closed under input related-reductions.

Applying this to our example, computing the input related reductions from nodes labelled 2,6 and 9 adds new state nodes labelled 3,5, and 7 to U_2.

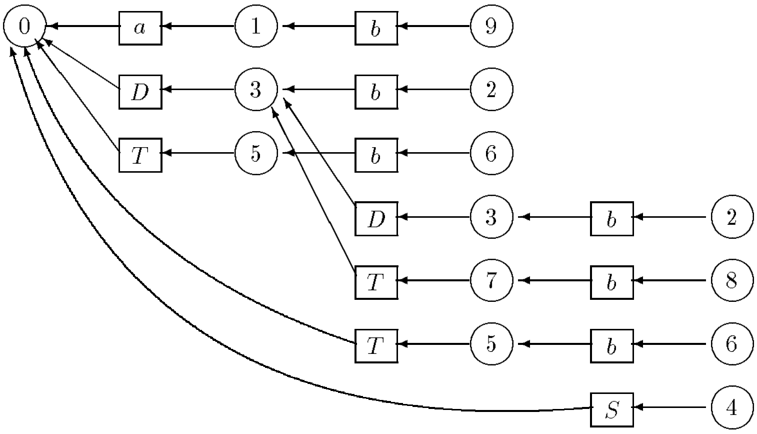


The newly created node labelled 7 has a reduction which leads back to state 0 and then to state 5. However, there is already a state node labelled 5 in U_2 which has a path of length 2 back to the state node in U_0 labelled 0, so we don't need to add anything to the GSS. The set U_2 is now closed under input related reductions, and step two is complete.

Reading the final input symbol, b, yields the following GSS



The only state which has a reduction on the lookahead symbol \$ is state 6. Performing the input related reduction on this state gives the final GSS below, in which $U_0 = \{0\}$, $U_1 = \{1, 3, 5\}$, $U_2 = \{2, 3, 5, 7, 9\}$ and $U_3 = \{2, 4, 6, 8\}$.



In general, if at any step the set U_i is empty, i.e. there are no transitions labelled a_i from any state node in U_{i-1}, or if U_n does not contain the accepting node of the DFA, then the input string is not in the language and the process stops and reports failure. Otherwise the process stops when the construction of U_n is complete and reports success. In our running example, since U_3 contains a state node labelled 4, the DFA accepting state, the input abb has been correctly recognised.

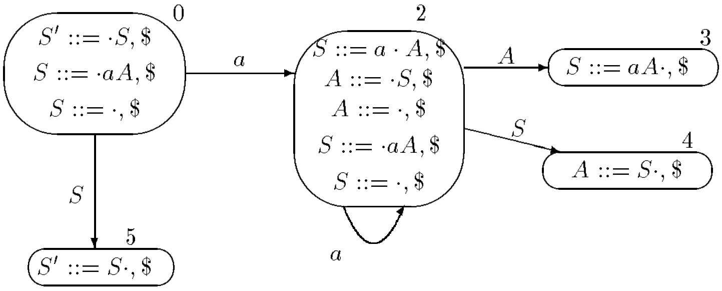
2.3 Dealing with ϵ-rules

For general grammars it is possible for the start state of the DFA, and of course other states, to contain reductions of the form $(A ::= \cdot, a)$. It is not difficult to incorporate such reductions into the model described above.

If a node $v \in U_i$ has label h, and h contains an item $(A ::= \cdot, a_{i+1})$ then we find the entry gk in row h, column A of the LR(1) table. We then check to see if there is a node $w \in U_i$ with label k. If not, we create one. We then create a node u labelled A and make u a predecessor of v and a successor of w. The effect of this is to add paths between two nodes at the same level in the GSS. In some cases this will result in cycles in the graph, but this does not affect the parsing process.

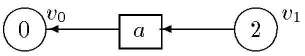
We illustrate this using the following example:

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

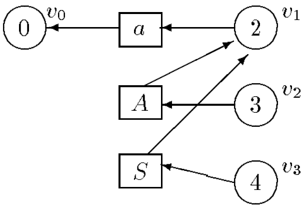


input: a

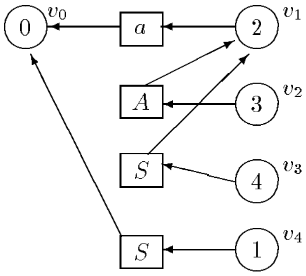
We begin in the start state, state 0, and construct a state node, v_0, labelled 0 in the GSS. There are no valid reductions in state 0 when the next input symbol is a, thus $U_0 = \{v_0\}$. We then read the a, create new nodes u_1 and v_2 labelled a and 2, respectively, and add v_1 to U_1.



There are two valid reductions in state 2, $(A ::= \cdot, \$)$ and $(S ::= \cdot, \$)$. These generate two new state nodes, v_2 and v_3, which are added to U_1 together with the paths of length 2 from them to v_1.



The reduction at v_2 results in the creation of a new node, v_4 labelled 1, in U_i. The valid reduction at v_3 results in a path of length 2 from v_1 to v_2, but this path already exists, so no further action is taken.



The construction of U_1 is now complete, and it contains a node whose label is the accept state of the DFA, so the input string is accepted.

2.4 Further issues to be considered

All of the algorithms that we shall discuss in this report are based on the process that we have described in the previous sections: begin with the start state of the DFA and construct its input related reduction-closure U_0. At step i with input a_i form the set $U_{i-1}a_i$ of states which can be reached from a state in U_{i-1} along a transition labelled a_i. Then construct the input related reduction-closure, U_i, of $U_{i-1}a_i$.

The issues which remain to be addressed are the explicit method by which the input related reduction-closures are computed (how the reductions are identified) and the nature and method of construction of the derivation trees for the given input string. It is useful to identify three aspects of the process

- ◇ the GSS and its construction
- ◇ the output derivation trees
- ◇ the method of constructing the output from the GSS

We can measure the efficiency of the overall parser in terms of the size of the GSS, the efficiency of the GSS construction method, the size of the output trees and the efficiency of the method by which these trees are constructed.

We shall look at the specific GSS construction algorithms given by Tomita [Tom86] and Farshi [NF91], and then we shall then give our own algorithm which is based on the insights gained by studying the underlying principles of Tomita's algorithm.

3 Tomita's algorithms

In his 1986 paper [Tom86] Tomita actually gives five algorithms for generalised LR parsing. All of these algorithms essentially calculate the input related reduction-closure of the current set $U_{i-1}a_i$ of states as described above. We shall refer to these algorithms as Algorithms 0, 1, 2, 3, and 4.

Algorithm 0 is just an introduction to the basic ideas an applies only to LR(1) grammars. We shall not discuss this algorithm.

Algorithm 1 contains most of the machinery needed for the recogniser role of the parser, but it is only applied to ϵ-free grammars and there is no direct discussion of the tree building role of the parser. This is the algorithm that we shall focus most of our attention on and the one we have been referring to as Tomita's first stage algorithm.

Algorithm 2 is an attempt to generalise Algorithm 1 so that it is applicable to all context free grammars, however the algorithm actually fails to terminate on grammars which contain hidden left recursion.

Algorithm 3 is a minor extension of Algorithm 2 which constructs a slightly more efficient GSS. The main aim of this extension is to allow the construction of more efficiently packed derivation trees.

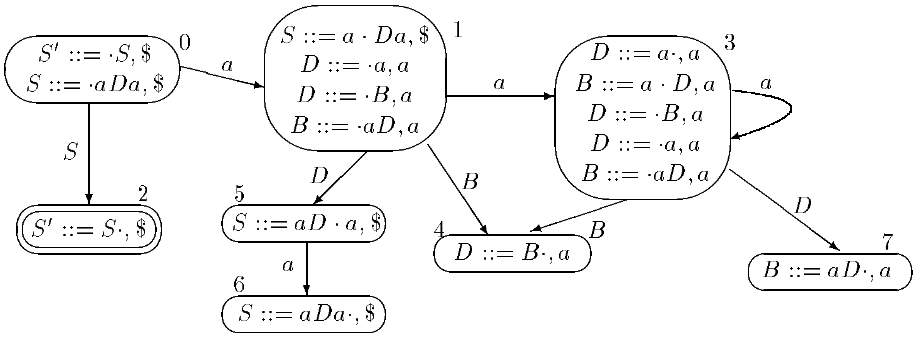
Algorithm 4 builds the same GSS by the same method as Algorithm 3 but it also has the mechanisms for building the output derivation trees in the form of a packed shared parse forest.

In this report we shall discuss Algorithm 1 with the modifications given in Algorithm 3. We shall then discuss the applicability of this algorithm to grammars which contain ϵ.

3.1 Processing reductions

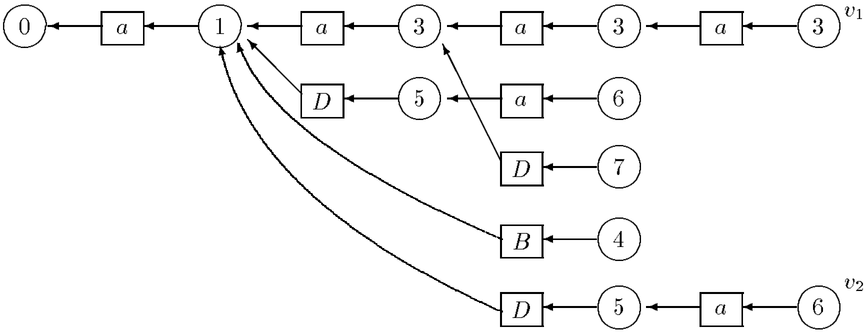
Consider a GSS whose associate input string is $a_1 \ldots a_n$. As we have already stated, each set, U_i, of level i state nodes in the GSS must be input related reduction-closed. This means that when a new state node v, labelled h say, is added to U_i we need to construct a (possibly empty) set of pairs of the form (A, m), one for each item of the form $(A ::= x_1 \ldots x_m \cdot, a_{i+1})$ in h. We then need to find all state nodes u in the GSS such that there is a path of length $2m$ from v to u. Thus we need to explore all paths of length $2m$ from v. The problem is that we may add a new successor node from v at a later stage in the process, creating a new path of length $2m$ from v which was not there when the valid reductions from v were originally dealt with. Since the set U_i must be input related reduction-closed, the node v must be processed again to deal with the new reduction. To illustrate this issue let us consider the following example.

$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & aDa \\ T & ::= & B \mid a \\ D & ::= & aD \end{array}$$

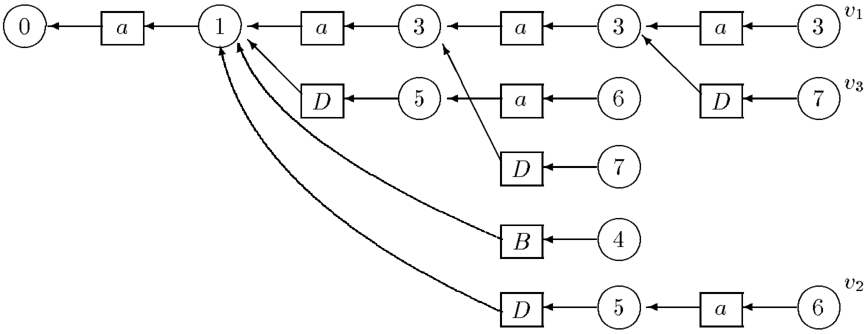


input: $aaaaa\$$

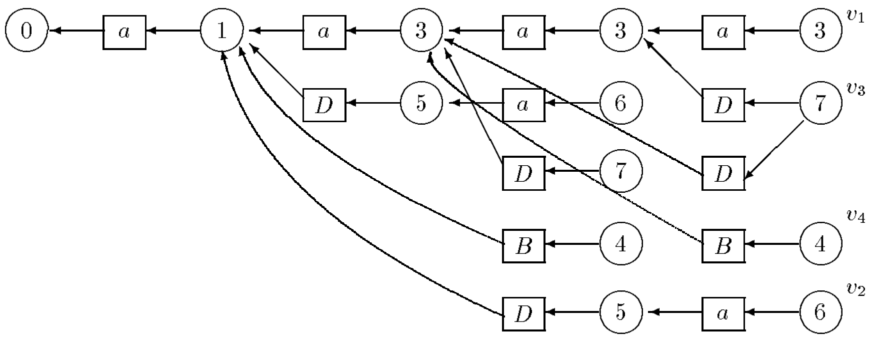
The following is the GSS which will be constructed when input $aaaa$ has been read and the set $U_3a = \{v_1, v_2\}$ has been built.



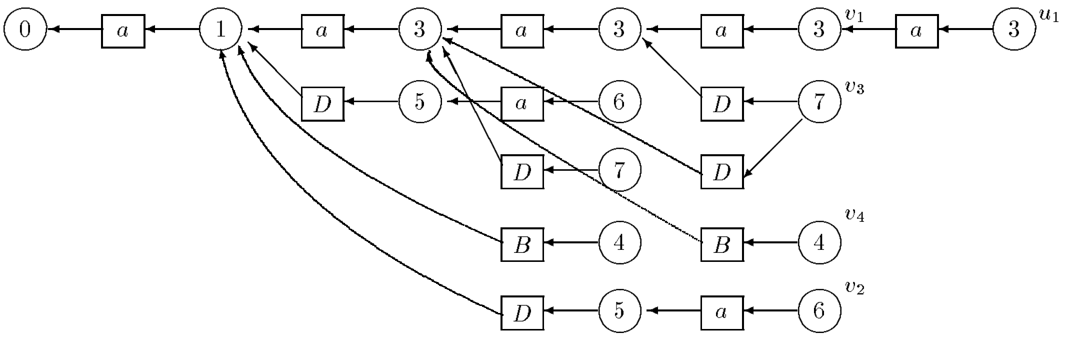
The node v_1 has a valid reduction of length 1 which results in the creation of a state node, v_3, labelled 7. The node v_2 has no valid reductions, the nodes v_1 and v_2 have been dealt with, and the GSS has the form



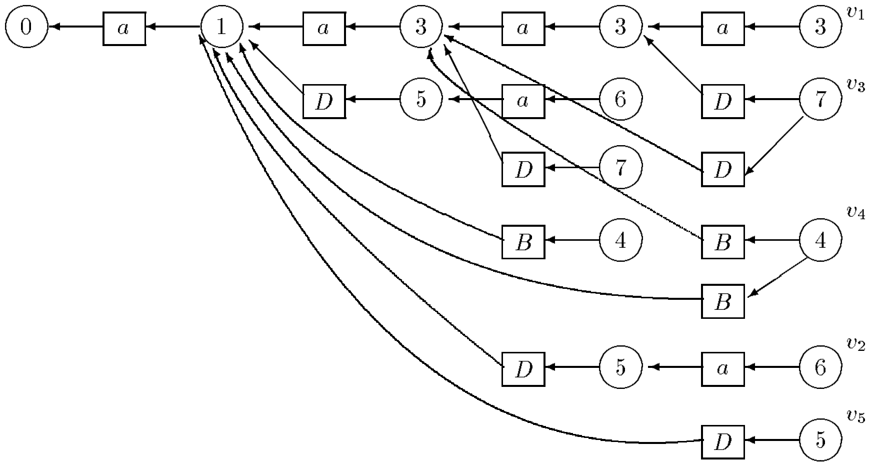
The node v_3 has a valid reduction of length 2 which results in the creation of a new state node v_4 labelled 4. There is only one path of length 2 from v_3 so this node has now been dealt with. There is also a valid reduction of length 1 in node v_4. There is already a state node in U_4 labelled 7, v_3, so no new node is created, the node v_3 is re-used.



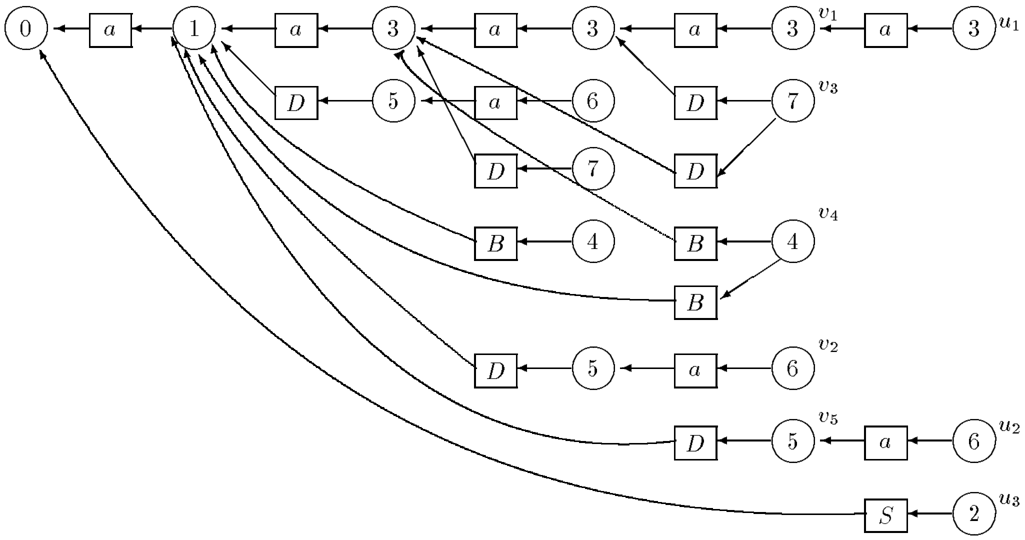
If we don't re-process v_3 then we will miss the valid reduction along the path of length 2 from v_3 to the node at level 2. In addition, when we then read the final input we end up with the following GSS in which $U_5 = \{u_1\}$ and the parser will (incorrectly) report that the input string is not derivable from the grammar.



However, if we re-process the node v_3 we will generate a new path from the node v_4. This in turn introduces a new valid reduction from v_4 which results in the construction of a new node, v_5, labelled 5.



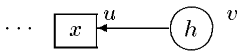
Now when we read the final input symbol and form the input related reduction on u_2 we get the following GSS



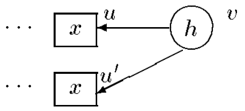
Then $U_5 = \{u_1, u_2, u_3\}$ contains a node, u_3, whose label is the accepting state of the DFA, so the input has been correctly recognised.

3.2 Storing pending reductions

If we need to visit a node again because a new path has been introduced we don't wish to search all paths again. Re-computing previously computed reductions will not cause an error because ultimately the check for the pre-existence of a path of length 2 will prevent duplication of the reductions. (If this check were not there then the process could fail to terminate!) However, graph searching is a relatively expensive operation and to repeat searches that have already been carried out will make the parser very inefficient. Tomita addresses this issue by keeping a set of pending reductions which records both the reduction rule to be applied and the first edge of the path(s) along which the reduction should be traced. When a state node is created in the GSS it has a label, h say, and a successor node, u, which is a symbol node.



The list of grammar rules is numbered and, for each item $(A ::= x_1 \dots x_m \cdot, a_{i+1})$ in h, where a_{i+1} is the next input symbol, the pair (rj, u) is added to the set of pending reductions, where j is the number of the rule $A ::= x_1 \dots x_m$. When the pair (rj, u) is processed, all paths from u of length $2m - 1$ are found. (This, of course, is equivalent to finding all paths of length $2m$ from v which include the edge (v, u).) If, at a later stage, a new path from v is created by adding a new edge whose source is v



then, for all reductions rj in h, the pairs (rj, u') are added to the set of pending reductions. This allows the new possibilities to be processed without reprocessing reductions along paths that have already been explored. In addition, Tomita maintains a set Q of pairs (v, h) where v is a state node in U_i with label k and h is a DFA state such that there is a transition labelled a_{i+1} from k to h. The set Q contains the 'shift actions' which must be used to construct the set $U_i a_{i+1}$ from the set U_i.

3.3 Tomita's first stage algorithm

We are now in a position to give Tomita's Algorithm 1. The algorithm uses the following sets for 'book keeping' purposes:

A: the set of state nodes in the GSS awaiting processing
U_i: the set of level i state nodes in the GSS
R: the set of pending reductions
Q: the set of pending shifts

There are three functions which build the GSS:
ACTOR: Processes state nodes from the A, putting pending shifts and reductions into the sets R and Q.
REDUCER: Processes pending reductions from the set R, creating new state and symbol nodes in the GSS as necessary. This function constructs the input related reduction-closures of the set $U_i a_{i+1}$.
SHIFTER: Processes the shifts from the set Q, constructing the state nodes in set $U_i a_{i+1}$.

Algorithm 1

input: an ϵ-free context free grammar whose production rules are uniquely numbered, a DFA constructed from this grammar in the form of a standard action/goto table, and an input string $a_1 \dots a_n\$$.

```
create a state node $v_0$ labelled with the start state 0 of the DFA.
set $U_0 = \{v_0\}$, $A = \emptyset$, $R = \emptyset$, $Q = \emptyset$
**for** $i = 0$ to $n$ do PARSE_SYMBOL($i$)

PARSE SYMBOL($i$) {
  $A = U_i$
  $U_{i+1} = \emptyset$
  **while** $A \neq \emptyset$ or $R \neq \emptyset$ **do**
    **if** $A \neq \emptyset$ do ACTOR($i$) **else** do REDUCER($i$)
  do SHIFTER(i)
}

ACTOR($i$) {
  remove $v$ from $A$, and let $h$ be the label of $v$
  **if** 'shift $k$' is an action in position $(h, a_{i+1})$ of the DFA table, add $(v, k)$ to $Q$
  **for** each entry 'reduce $j$' in position $(h, a_{i+1})$ of the DFA table {
    **for** each successor node $u$ of $v$ add $(u, j)$ to $R$ }
}

REDUCER($i$) {
  remove $(u, j)$ from $R$
  let $m$ be the length of the right hand side of rule $j$ and let $X$ be
            the symbol on the left hand side of rule $j$
  **for** each state node $w$ which can be
            reached from $u$ along a path of length $(2m - 1)$  **do** {
    let $k$ be the label of $w$ and let $gl$ be the entry in
                                  position $(k, X)$ of the DFA table
    **if** there is no node in $U_i$ labelled $l$ then create a new
              state node, $v$, in the GSS labelled $l$ and add $v$ to $U_i$ and to $A$
    let $v$ be the node in $U_i$ labelled $l$
    **if** there is a path of length 2 in the GSS from $v$ to $w$ then do nothing
    **else** {
        create a new symbol node $u'$ in the GSS labelled $X$
        make $u'$ a successor of $v$ and a predecessor of $w$
        **if** $v$ is not in $A$ {
          **for** all reductions $rk$ in position $(l, a_{i+1})$ of the table add $(u, k)$ to $R$}
  }
}

SHIFTER($i$) {
  **while** $Q \neq \emptyset$   **do**  {
    remove an element $(v, k)$ from $Q$
    **if** there is no node, $w$, labelled $k$ in $U_{i+1}$ create one
    **if** $w$ does not have a successor node, $u$, labelled $a_{i+1}$, create one
    **if** $u$ is not already a predecessor of $v$, make it one  }
}

let $q$ be the accepting state of the DFA
**if** $U_n$ contains a state whose label is $q$ report success  **else** report failure
```

Note: The action of the SHIFTER which results in the sharing of the symbol node in the case where the shift results in the parser moving to the same state, is actually a modification that Tomita introduces in his third stage algorithm, however we have included it here because it is only a minor extension and we have included in our implementation of Algorithm 1e below.

4 Tomita's algorithm applied to general grammars

There is no operational aspect of Tomita's Algorithm 1 which prevents it from being applied to grammars which include ϵ-productions. We can modify Algorithm 1 slightly to apply ϵ-reductions once, when a node is processed for the first time. The effect of this is that paths will be added between nodes in the same U_i in the case of nullable production rules. (Tomita's Algorithm 2 contains this modification along with other modifications which result in the splitting up of the U_i into subsets. We shall briefly discuss this later.) We shall call the algorithm Algorithm 1e to indicate that it is admitting ϵ-productions.

In this section we shall give Algorithm 1e and show that it works on an example grammar which contains hidden left recursion. We shall the consider the problems with this algorithm and identify the grammar properties which can trigger these problems.

4.1 Algorithm 1e

The following is a slight modification of Tomita's Algorithm 1 which allows input grammars to contain productions.

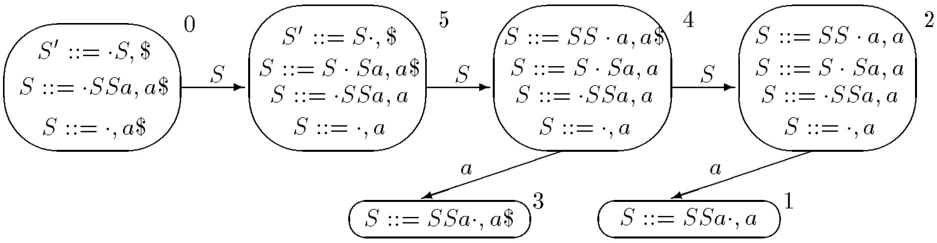
input: a context free grammar whose production rules are uniquely numbered, a DFA constructed from this grammar in the form of a standard action/goto table, and an input string $a_1 \dots a_n\$$.

```
create a state node $v_0$ labelled with the start state 0 of the DFA.
set $U_0 = \{v_0\}$, $A = \emptyset$, $R = \emptyset$, $Q = \emptyset$
for $i = 0$ to $n$ do PARSE_SYMBOL($i$)

PARSE SYMBOL($i$) {
  $A = U_i$
  $U_{i+1} = \emptyset$
  while $A \neq \emptyset$ or $R \neq \emptyset$ do
    if $A \neq \emptyset$ do ACTOR($i$) else do REDUCER($i$)
  do SHIFTER(i)
}

ACTOR($i$) {
  remove $v$ from $A$, and let $h$ be the label of $v$
  if 'shift $k$' is an action in position $(h, a_{i+1})$ of the DFA table add $(v, k)$ to $Q$
  for each entry 'reduce $j$' in position $(h, a_{i+1})$ of the DFA table
       if the length of $j$ is 0 add $(v, j)$ to $R$
       else add $(u, j)$ to $R$, for each successor node $u$ of $v$
}

REDUCER($i$) {
  remove $(u, j)$ from $R$
  let $m$ be the length of the right hand side of rule $j$ and let $X$ be
            the symbol on the left hand side of rule $j$
  if $m = 0$
    let $k$ be the label of $u$ and let $gl$ be the entry in position $(k, X)$
                              of the DFA table
    if there is no node in $U_i$ labelled $l$ then create a new state node, $v$,
          in the GSS, labelled $l$ and add $v$ to $U_i$ and to $A$
    let $v$ be the node in $U_i$ labelled $l$
    if there is a path of length 2 in the GSS from $v$ to $u$ then do nothing
     else {
         create a new symbol node $u'$ in the GSS labelled $X$ and make
                $u'$ a successor of $v$ and a predecessor of $u$
          if $v$ is not in $A$, for all reductions $rk$ in position $(l, a_{j+1})$
            of the DFA table, with length$\neq$ 0, add $(u', k)$ to $R$ }
  else
    for each state node $w$ which can be reached
                  from $u$ along a path of length $2m - 1$  do {
      let $k$ be the label of $w$ and let $gl$ be the entry in position $(k, X)$ of the table
      if there is no node in $U_i$ labelled $l$ then create a new
              state node, $v$, in the GSS labelled $l$ and add $v$ to $U_i$ and to $A$
      let $v$ be the node in $U_i$ labelled $l$
      if there is a path of length 2 in the GSS from $v$ to $w$ then do nothing
      else {
           create a new symbol node $u'$ in the GSS labelled $X$
           make $u'$ a successor of $v$ and a predecessor of $w$
           if $v$ is not in $A$ {
             for all reductions $rk$ in position $(l, a_{i+1})$ of the table add $(u, k)$ to $R$}
    }
}

SHIFTER($i$) {
  while $Q \neq \emptyset$   do  {
    remove an element $(v, k)$ from $Q$
    if there is no node, $w$, labelled $k$ in $U_{i+1}$ create one
    if $w$ does not have a successor node, $u$, labelled $a_{i+1}$, create one
    if $u$ is not already a predecessor of $v$, make it one }
}

let $q$ be the accepting state of the DFA
if $U_n$ contains a state whose label is $q$ report success  else report failure
```

4.2 An example with hidden left recursion

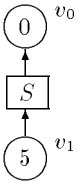
We illustrate Algorithm 1e by running it on the following grammar and input string aa (note the grammar contains hidden left recursion).

$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & SSa \mid \epsilon \end{array}$$



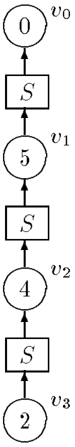
	\$	a	S
0	r2	r2	g5
1		r1	
2		r2/s1	g2
3	r1	r1	
4		r2/s3	g2
5	acc	r2	g4

We begin in state 0 with just a node v_0 labelled 0 in the graph structured stack. So we have $A = U_0 = \{v_0\}$, $R = Q = \emptyset$ and the lookahead symbol is $a_1 = a$. We remove v_0 from A and note that the only action is to reduce using the rule $S ::= \epsilon$. Since the length of the rule is 0, this adds the pending reduction $(v_0, 2)$ to R. When we process this reduction we find that the goto node is state 5. There is no node labelled 5 in U_0 so we create one, v_1, and add a path from it to v_0 via a node labelled S. The node v_1 is added to U_0 and to A.



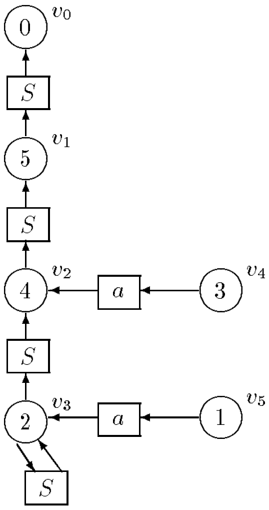
$A = \{v_1\}$

We then remove the node v_1 from A, process the reduction $S ::= \epsilon$, create an new node, v_2 labelled 4, and a path from v_2 to v_1. Processing v_2 creates another new node, v_3 labelled 2, and adds the pending shift $(v_2, 3)$ to the list Q.



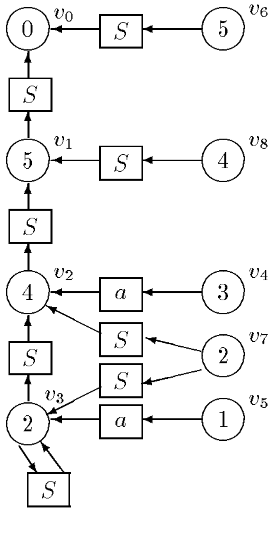
$R = \emptyset$
$Q = \{(v_2, 3)\}$
$A = \{v_3\}$

Processing v_3, there is already a node in U_0 labelled 2, v_3 itself. Thus we simply add a path from v_3 to itself and add the pending shift $(v_3, 1)$ to Q. Since there are no reductions in state 2 of length greater than zero, the addition of a new edge from node v_3 does not create any new pending reductions. So we perform the pending shifts, creating new states v_4 and v_5 which are added to U_1.



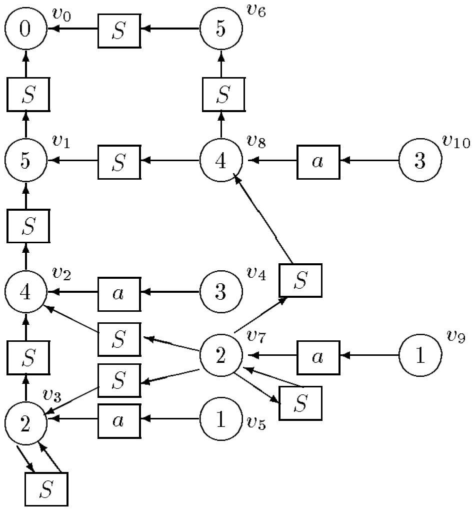
$A = \{v_4, v_5\}$

Now we add the nodes v_4, v_5 to A and remove and process v_4. From state 3 we can apply the reduction $S ::= SSa$ which takes us back to v_0 and hence the goto state is 5. There is no node labelled 5 in U_1 so one, v_6, is created and added to A. We then remove and process v_5. There are three paths of length 6 from v_5, which take us to nodes v_3, v_2 and v_1. The first two generate a new node v_7, labelled 2, and paths of length 2 from v_7 to v_3 and v_2, and the third generates a new node v_8, labelled 4, and a path of length 2 from v_8 to v_1.



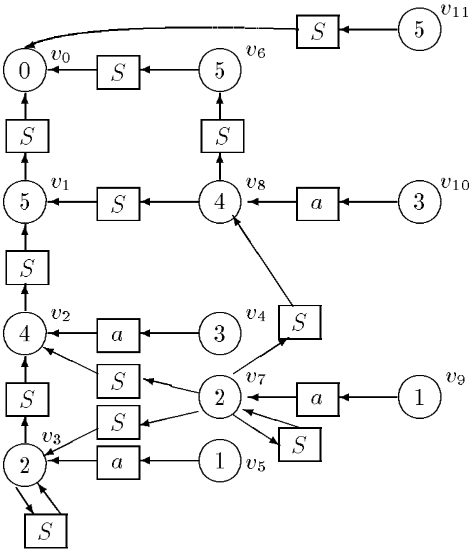
$A = \{v_6, v_7, v_8\}$

We remove and process node v_6 which results in the creation of a path of length 2 from v_8 to v_6. We then remove and process v_7 which results in a new path of length 2 from v_7 to itself, and in a pending shift $(v_7, 1)$. Finally we remove and process v_8 which results in a new path of length 2 from v_7 to v_8 and a pending shift $(v_8, 3)$. Since state 2 contains only reductions of length 0, no new reductions are created down the new edge, and the construction of U_1 is complete. The pending shifts are then processed, creating new nodes v_9, v_{10} which are added to U_2 and to A.



$A = \{v_9, v_{10}\}$

Removing v_9 from A, there are no actions that can be applied. Processing v_{10} results in the creation of a new node, v_{11}, labelled 5 and a path of length 2 from v_{11} to v_0. The second path of length 6 from v_{10} also results in need for a node labelled 5, but there already is one, and a path of length 2 of the required type, so no further action is taken. There are no actions which can be applied from state 5 so the construction process stops.



Since U_2 contains a node, v_{11}, whose label is the accepting state of the DFA, the algorithm reports success and the string aa is accepted.

4.3 The (in)correctness of Algorithm 1e

Tomita's Algorithm 1 and Algorithm 1e will always terminate, even given a grammar with hidden left recursion, because each U_i can only contain as many nodes as there are states in the original DFA and there is at most one path of length 2 from any node in any U_i to any node in any U_j.

The problem is that it is possible for Algorithm 1e to reject strings which are actually in the language in certain special cases. We shall now look at these special cases.

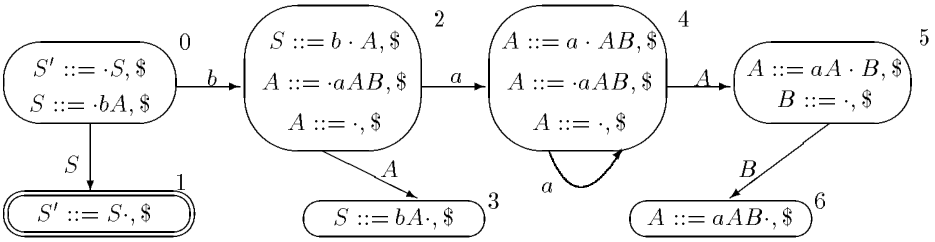
4.4 Hidden right recursion

In Tomita's Algorithm 1 and Algorithm 1e reductions are added to R with the first edge down which the reduction is to be applied. This prevents the algorithm from performing unnecessary work in re-tracing reductions down paths that it has already explored. However, as a consequence it is necessary to ensure that new paths from an existing node in the GSS are only created by the addition of a new edge from the first node in the path. We now illustrate this point.

We say that a non-terminal, A, has *hidden right recursion* if it has a rule of the form $A ::= \alpha A\beta$, where $\beta\overset{+}{\Rightarrow}\epsilon$. A grammar has hidden right recursion if it has a non-terminal with hidden right recursion.

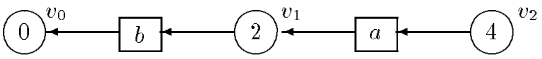
Consider the following example.

$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & bA \\ A & ::= & aAB \mid \epsilon \\ B & ::= & \epsilon \end{array}$$



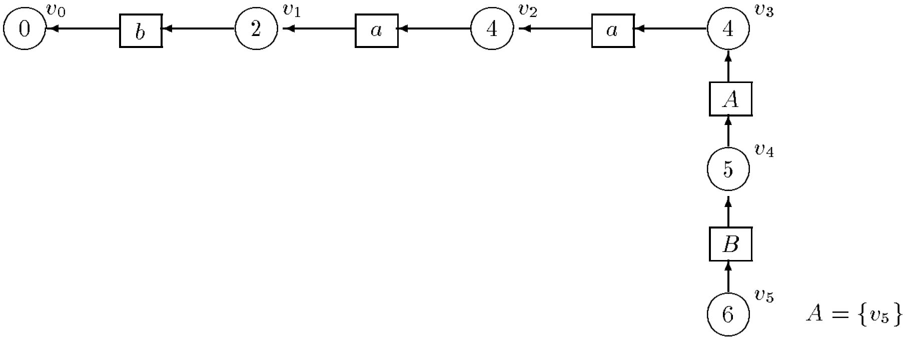
	$	a	b	A	B	S
0			s2			g1
1	acc					
2	r3	s4		g3		
3	r1					
4	r3	s4		g5		
5	r4				g6	
6	r2					

We run Algorithm 1e with the above table and input string baa. There are no reductions on lookahead b or a so the construction of U_0, U_1 and U_2 is straightforward and generates the GSS

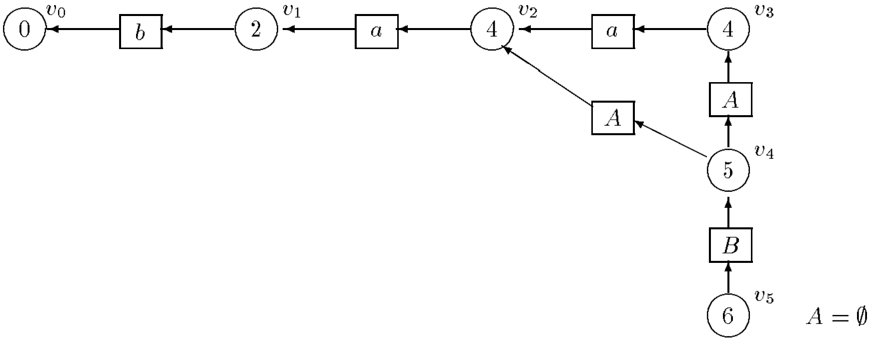


$A = \{v_2\}$

We then process the pending shift from v_2 on the second a, to create a new node, v_3, labelled 4 which is added to U_3 and then to A. This is the only node in A so we remove and process it. The action $r4$ creates a new node, v_4, labelled 5 which is added to U_3 and A, and a path of length 2 from v_4 to v_3. Again, v_4 is the only node in A so we remove and process it. The action $r4$ creates a new node, v_5, labelled 6 which is added to U_3 and A, and a path of length 2 from v_5 to v_4.

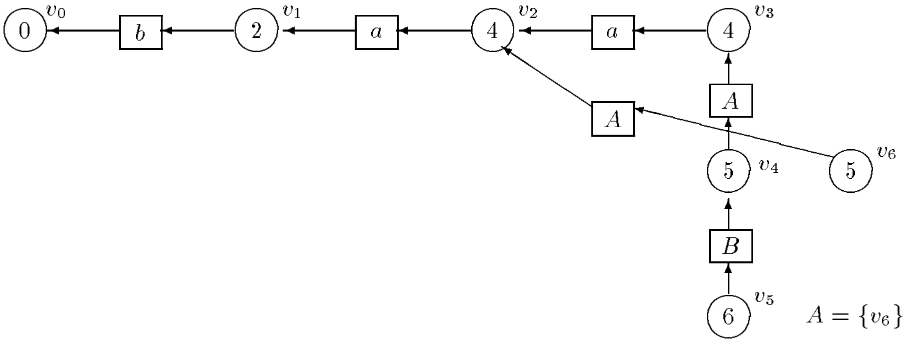


Now we remove and process v_5. There is a reduction of length 3 which takes us back to the node v_2 and then, from row 4 of the table, we see that we need to add a path of length 2 from the node, v_4, in U_3 which has label 5 to the node v_2.

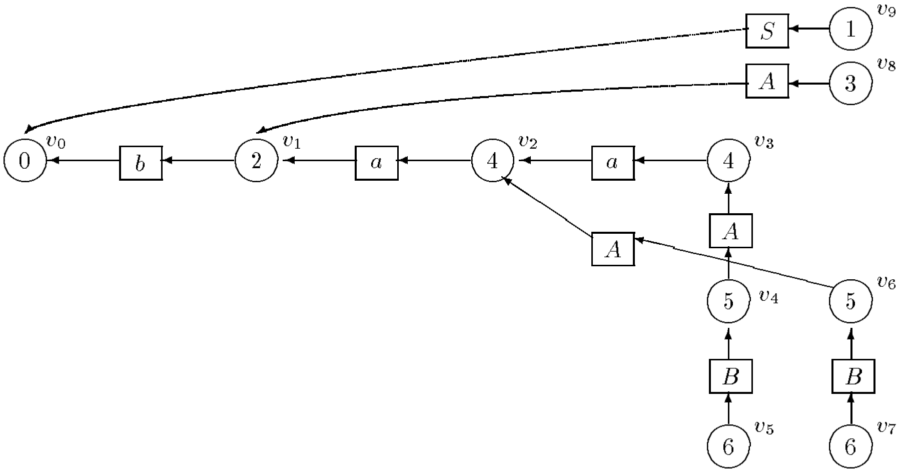


Algorithm 1e now requires us to add any reductions, of length greater than 0, in position $(5, \$)$ of the table to the pending reduction set R. There are no such reductions so R and A are empty and the algorithm terminates. Since there is no node in U_3 with label 1 (the accept state of the DFA) the string baa is (incorrectly) rejected.

The problem is that a new path has also been created from node v_5 and we need to apply a reduction down this path. Tomita addresses this issue by subdividing the sets U_i and creating a new subset every time a reduction is applied. The effect of this is that when the first reduction from v_5 is computed a new subset $U_{3,1}$ is created. The existing node, v_4 is not in this subset so a new node, v_6, labelled 5 is constructed.



Then, when v_6 is processed, the reduction is applied to construct a new node labelled 6, and then finally the reduction from state 6 is applied.



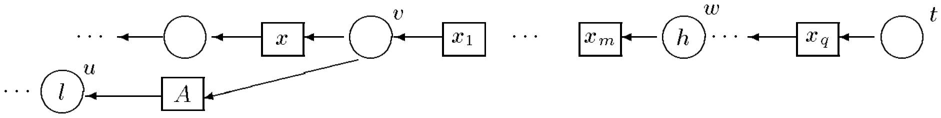
This approach, which is the basis of Tomita's Algorithm 2 although slightly simplified in the above description, will result in the correct recognition of the string baa but the introduction of the subdivisions of the U_i causes the algorithm to fail to terminate when given a grammar with hidden left recursion.

Farshi addresses this issue by revisiting all the nodes in U_i and finding all paths which contain the newly added edge and then applying all reductions down these paths. We shall formally describe this method in Section 5, but the method is inefficient in that it has to search for all paths containing a specified edge. Thus, before describing Farshi's method we shall analyse the grammars on which Algorithm 1e fails. We shall use this information in Section 6 to give a different solution which involves modifying the LR(1) DFA so that Algorithm 1e works correctly on all grammars.

4.5 Right nullable rules

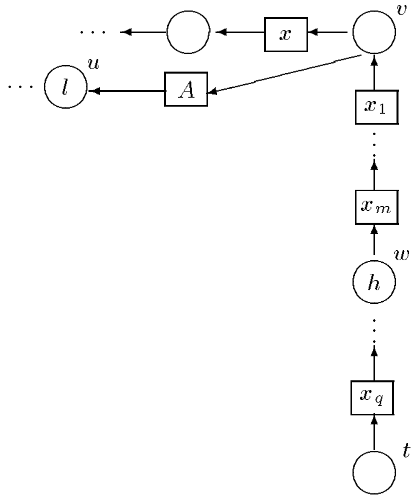
In the previous section we saw that Algorithm 1e incorrectly rejected a string as a result of failing to apply a reduction down a path which was created after the node which generated the reduction was processed. This occurred because a new path was created by adding a new edge from a node which was in the middle of an existing path, i.e. from a node which already has at least one parent. In this section we analyse the grammar properties which permit this situation to occur.

Suppose that we are at step i of the algorithm, so that the set U_i is being constructed. Suppose also that we have an existing path in the GSS whose first node is t, and suppose that a new edge is created from a node, v say, in the path to a node u. Since the node v is assumed to already exist, this path must have been created as the result of a reduction, $A ::= \alpha$ say.



This new path will only create a problem if there is a node, w say, which has already been processed, which is not equal to v, and whose state, h say, contains a reduction, $B ::= \beta\cdot$ say, where twice the length of β is at least the length of the new path from w to u. Also, since $v \neq w$, we may assume that $m \geq 1$.

First we note that a new edge is only added from a state node in the GSS if that node belongs to the current U_i, i.e. if the node was created during the current step of the algorithm. Thus v must be in U_i. All the nodes, s, for which there is a path from s to v must have been created after v and hence must also be in U_i. Thus we re-draw the above GSS fragment as follows:

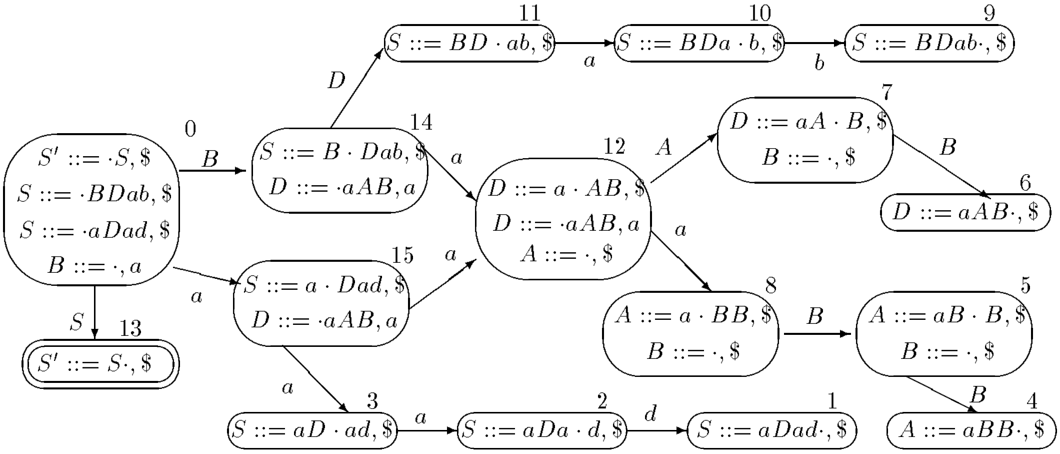


It is not hard to show, by induction on the order on which the edges were created, that if a path of length 2 is created between any two nodes in the same U_j then this path must have been created as a result of a reduction of the form $C ::= \gamma$ where $\gamma \overset{*}{\Rightarrow} \epsilon$, see Lemma 3 below. Thus we may assume that for each of the x_j which label symbol nodes on the path from t to v we have $x_i ::= \gamma_i$ where $\gamma_i \overset{*}{\Rightarrow} \epsilon$.

We say that a rule of the form $A ::= \alpha\beta$ where $\beta \overset{+}{\Rightarrow} \epsilon$ is *right nullable* and a grammar is right nullable if it contains a right nullable rule.

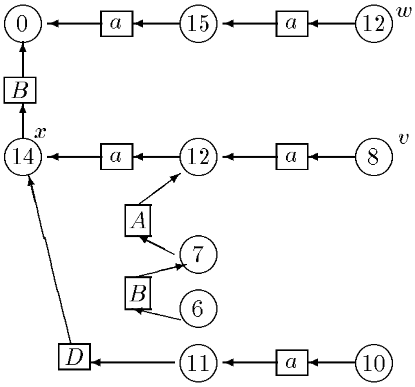
In order for a reduction to fail to be applied it is necessary both for an edge to be added part of the way down an existing path and for the reduction which adds this edge to be processed before the reduction which is to be applied down the new path. In other words, our addition of ϵ-handling capability to Tomita's first algorithm makes that algorithm sensitive to the order in which reductions are processed. Consider the following example,

$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & BDab \mid aDad \\ D & ::= & aAB \\ A & ::= & aBB \mid \epsilon \\ B & ::= & \epsilon \end{array}$$

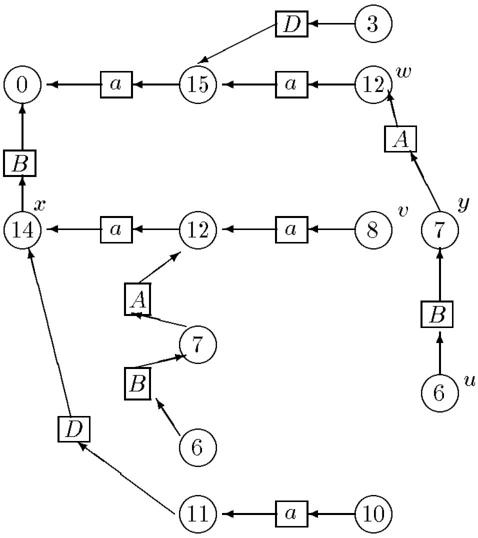


	$	a	b	d	A	B	D	S
0		s15/r6				g14		g13
1	r2							
2				s1				
3		s2						
4		r4						
5		r6				g4		
6		r3						
7		r6				g6		
8		r6				g5		
9	r1							
10			s9					
11		s10						
12		s8/r5			g7			
13	acc							
14		s12					g11	
15		s12					g3	

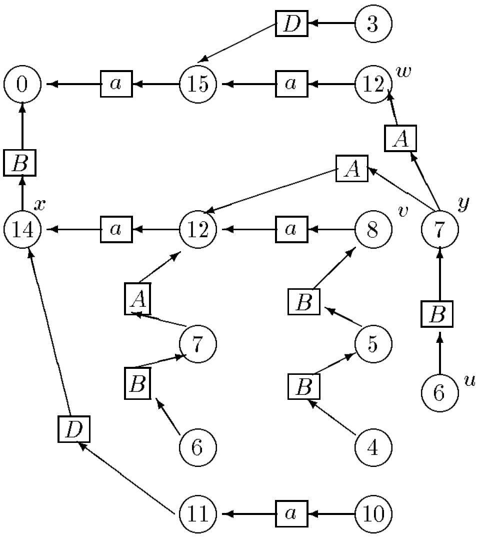
We construct the GSS for the input string *aaab*. The GSS constructed at the point where the second *a* is shifted is



If we process the node w first, and then the nodes which are created from this we get

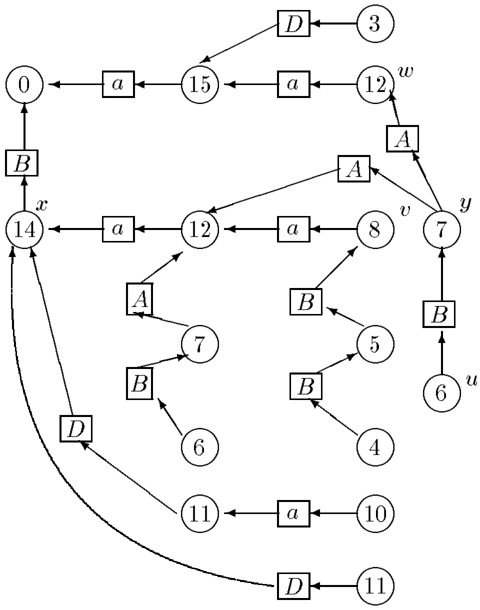


If we then process the node v and the subsequently created nodes we get

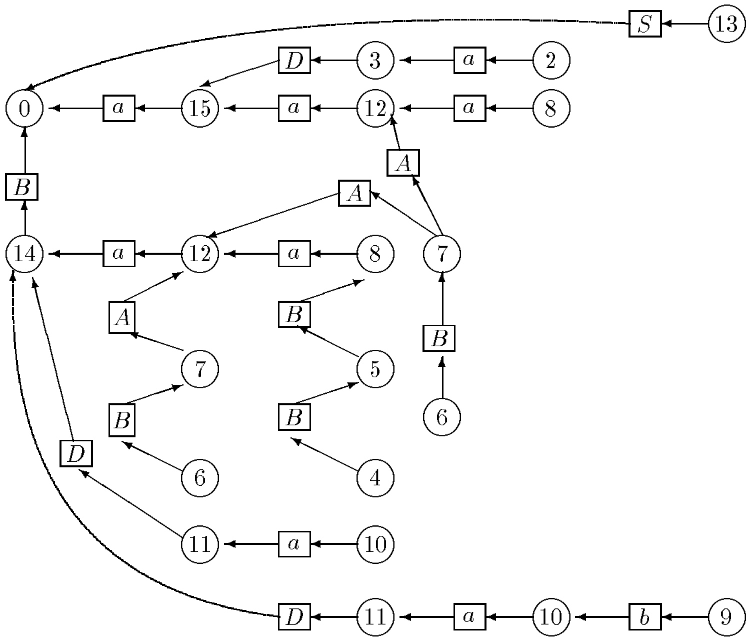


The subsequent reduction down a path of length 6 from node u to node x is not detected. This results in the ultimate failure of the parse.

If we had processed node v before node w, and the node w before the node y, then at the completion of the construction of U_2, i.e. just before the third a was shifted, we would have



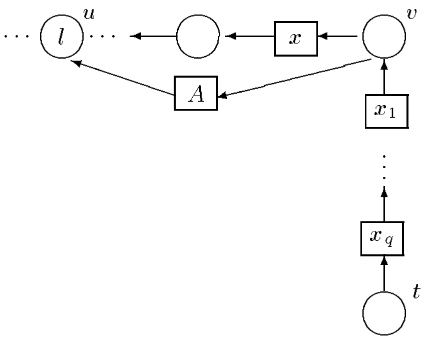
We can then successfully complete the parse, generating the GSS



Thus in this case we can avoid the problem of generating new paths down which a previously processed reduction must be applied by choosing the order in which nodes are processed carefully. (Note that this processing order will also result in a successful parse of the string $aaad$, although if y were to be processed before w

then this would not be the case.) However, it is not always the case that we can do this.

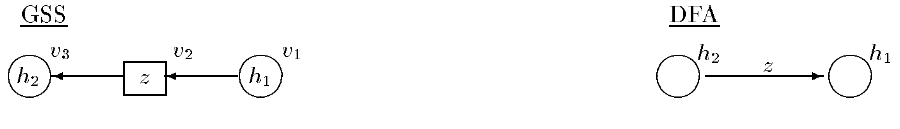
The most likely source of problems which cannot be avoided by processing reductions in an appropriate order are those cases when the additional edge is created as a result of applying a reduction down the path to which the new edge will be added. In other words, when a reduction is applied from the node t down to the node u and the new path is added from a node, v, between t and u, to u.



In this case it is not possible to construct the path $t, \dots, v, u$ before the node t is processed because the path from v to u is only created after t is processed. We shall now show that the only way that a new edge can be added to the middle of an existing path as a result of applying a reduction down that path is when the grammar contains hidden right recursion. Thus, in this sense, it is hidden right recursion which ultimately breaks Tomita's Algorithm 1 when it is applied to general context free grammars.

First we note that, by construction of the GSS, for the path from v to u to be created as a result of a reduction from t, t must contain an item of the form $(A ::= \dots xx_1 \dots x_q\cdot, a_{i+1})$ and, as we saw above, $x_l \overset{*}{\Rightarrow} \epsilon$, for $1 \leq l \leq q$.

It is not hard to see that if there is a path v_1, v_2, v_3 in the GSS where v_1 has label h_1, v_3 has label h_2 and v_2 has label z then there is a transition from h_2 to h_1 labelled z in the DFA.



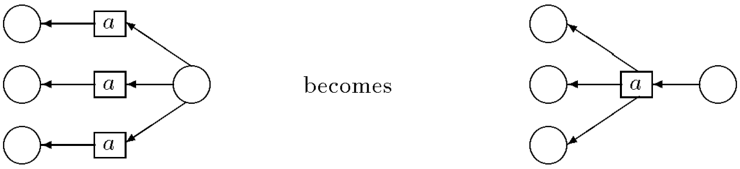
Since all successor nodes of a given state node in the GSS must have the same label, we must have $x = A$ in the GSS path that we are considering. So $A ::= \dots Ax_1 \dots x_q$, where $x_1 \dots x_q \overset{*}{\Rightarrow} \epsilon$, and hence A has hidden right recursion.

In Section 6 we give an algorithm based on an extension of the underlying DFA table in which right nullable rules are treated as reductions. We shall show that this algorithm generates correct parsers for all context free grammars.

4.6 Merging symbol nodes

We complete our discussion of Tomita's algorithms by considering one further modification that he makes.

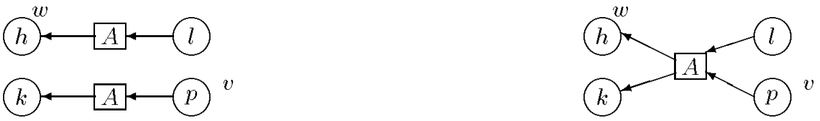
We have seen that we can reduce the number of symbol nodes in the GSS by sharing nodes labelled with a terminal,



Of course, in order to simply recognise whether or not a string is in the language we do not need to include the symbol nodes in the GSS at all. However, Tomita extends his algorithms so that they produce derivation trees and he constructs the GSS so that the symbol nodes in the GSS correspond exactly to nodes in the derivation trees. With this constraint in mind we consider under what circumstances non-terminal symbol nodes can be merged.

Two symbol nodes in the GSS can correspond to the same node in the derivation tree if they have the same symbol and if they generate the same portion of the input string. If a symbol node labelled A has parent node $v \in U_i$ and child node $u \in U_j$ then it generates the portion $a_{j+1} \ldots a_i$ of the input $a_1 \ldots a_m$. Thus to be candidates for merger two symbol nodes must have parent nodes in the same U_i and child nodes in the same U_j.

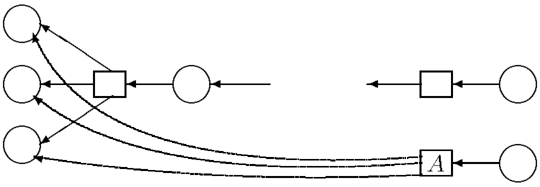
We must remember that the role of the GSS is to record paths taken through the DFA, and merging two symbol nodes which have different parents and different children could create spurious paths in the GSS. For example, merging the following two nodes will create a path from v to w which does not really exist.



We can avoid this by requiring that nodes are only shared if they have the same parent.

We have to check, when adding a new symbol node, whether there already exists a path of length 2 from the required state in U_i to some state in U_j, but this check is not much more effort than the check for the existence of a path of length 2 from the required state in U_i to the base state, which is already carried out. The problem is that, as for the case with right nullable rules, if we allow the reuse of symbol nodes in this way it is possible to add a new edge from the second node of an existing path, and hence to fail to perform reductions down the new path which is created.

Tomita addresses this issue by only allowing sharing of symbol nodes at the time that the node is created. This means that symbol nodes can only be shared if the derivation paths which created them co-incide down to the penultimate node.



This has the advantage of not requiring any checking to see whether paths already exist, but it severely constrains the number of symbol nodes that can be shared.

In our algorithm, see Section 6, we shall record pending reductions with the second state node down the path which they are to be applied, rather than with the first edge as Tomita does. This will allow us to share all symbol nodes with the same parent, and children in the same U_j, because reductions will be recorded with the newly added child rather than with the existing incoming edge. Before this we look at Farshi's method for identifying and storing pending reductions.

5 Farshi's algorithm

In this section we discuss Farshi's modification to Tomita's algorithm [NF91], which he designed to address the problem with hidden left recursion that Tomita's Algorithm 2 displayed. We quote the algorithm here verbatim from Farshi's paper because we wish to comment on its efficiency, particularly in light of the comment made on page 74 of [NF91] which states: "However, the new algorithm works exactly like the original one in case of grammars that have no ϵ-productions. This algorithm has no extra costs beyond that of the original algorithm."

5.1 Farshi's recognition algorithm for general grammars

input: a context free grammar G whose production rules are uniquely numbered, a DFA with start state s_0, constructed from this grammar, in the form of a standard action/goto table, and an input string $a_1 \ldots a_n\$$.

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

PARSEWORD($i$)
  $A := U_i$
  $R := \emptyset$; $Q := \emptyset$
  repeat
    if $A \neq \emptyset$ then do ACTOR  else  if $R \neq \emptyset$ then do COMPLETER
  until $R = \emptyset$ and $A = \emptyset$
  do SHIFTER

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

COMPLETER
  Remove an element $(w, p)$ from $R$
```

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

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

5.2 The efficiency of Farshi's algorithm

In this section we shall compare Farshi's algorithm with Tomita's algorithms. While we believe it is the case that Farshi's algorithm constructs the same GSS as Tomita's original algorithm for ϵ-free grammars, we do not believe that the algorithm as stated has 'no extra costs beyond that of the original algorithm'. In addition, Farshi's algorithm does not adopt the symbol node merging introduced in Tomita's Algorithm 3. We now discuss these issues with respect to the method Farshi uses for storing pending reductions.

Recall that in Tomita's algorithms pending reductions are stored with the first symbol node on the path down which they are to be applied. Thus, if a new path is created by adding an edge in the middle of an existing path, Tomita's method does not provide a mechanism for storing pending reductions down the new path without also re-exploring the other paths that begin with the same node. In order to overcome this, Farshi's algorithm stores reductions together with the nodes at the ends of the paths down which the reductions can be applied. This is essentially equivalent to Tomita's method, it is just that the path tracing is carried out a different point in the algorithm. (Although it is possible that with Farshi's method the set R of pending reductions will tend to be larger.)

It is trivial to modify Farshi's algorithm so that symbol nodes corresponding to terminals are merged where possible, and we have done this in the version that we have implemented, see Section 5.3. However, as reductions are stored with the final node in the path it is not possible easily to tell when two different reduction,node pairs share the same path up to the penultimate node. Thus the second type of symbol node merging employed in Tomita's Algorithm 3 is not used either in Farshi's algorithm or in our implementation of it.

The main issue with Farshi's algorithm is the additional effort introduced by the need to search for all paths containing a given node, z, when storing pending reductions. (This is the search carried out as part of the inner loop on line 12 of the COMPLETER function above.) As actually stated in the algorithm, it is necessary to search for all nodes on the current frontier, U_i, which are not awaiting processing, and then, for all reductions associated with these nodes, find all paths of the appropriate length which include z. This proposal essentially defeats the object of Tomita's original decision to store reductions with the first node on the appropriate path. If we just stored the reductions alone, and re-stored them each time an edge is added to the GSS, then the only additional cost over Farshi's method would be a check for the existence of path of length 2 in the case where the reduction had already been applied. (This check would confirm the existence of the path and no further work would be done.)

There is a comment in Farshi's algorithm a few lines above the searching loop which states that for non-ϵ grammars the 'for' loop only executes for $v = u$. It is true that this is the only case in which additional elements are added to R, but with the algorithm as stated, in all cases the searching will be carried out for all nodes in U_i not awaiting processing. We assume that Farshi intends the implementor to put in a check for non-ϵ grammars before the comment and to implement a different action in this case.

If the GSS is implemented so that searching back up paths is as efficient as searching down them, i.e. if the predecessors of a node can be found efficiently, then, with a somewhat more careful description of the search process, we can modify Farshi's algorithm so that it has essentially no additional costs over Algorithm 1e on grammars with no right nullable rules, and so that the searching costs for general grammars are reduced. Of course, implementing directed graphs so that predecessors can be found as efficiently as successors does increase the space required by the implementation. We assume that this is the reason that Farshi's algorithm does not adopt this approach. However, with modern memory availability we feel that the space cost is worth trading for increased speed of the algorithm. Thus we present our modified version of Farshi's algorithm in Section 5.3, and this version is the basis of the implementation that we have used in our practical comparison of the various algorithms, see Section 7. In Section 6 we shall present an algorithm in which pending reductions are stored with the first node on the new part of the path down which they are to be applied, thus avoiding any additional searching over that required in Tomita's Algorithm 1.

5.3 A modified version of Farshi's algorithm

In this section we give a modified version of Farshi's algorithm in which the costs involved with searching for paths containing a specified node are reduced (provided that the GSS implementation allows predecessor nodes to be efficiently found), and in which symbol nodes corresponding to terminals are merged in certain cases. The only changes are in the SHIFTER and the COMPLETER.

input: a context free grammar G whose production rules are uniquely numbered, a DFA with start state s_0, constructed from this grammar, in the form of a standard action/goto table, and an input string $a_1 \ldots a_n\$$.

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

PARSEWORD($i$)
  $A := U_i$
  $R := \emptyset$; $Q := \emptyset$
  repeat
     if $A \neq \emptyset$ then do ACTOR  else  if $R \neq \emptyset$ then do COMPLETER
  until $R = \emptyset$ and $A = \emptyset$
  do SHIFTER

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

```
COMPLETER
 Remove an element $(w, p)$ from $R$
 $N := LHS(p)$; $s :=$GOTO(STATE($w$), $N$)
 if there exists $u \in U_i$ such that STATE($u$)= $s$ then
     begin
       if there does not exist a path of length 2 from $u$ to $w$ then
         begin
            create a vertex $z$ labelled $N$ in $\Gamma$
            create two arcs in $\Gamma$ from $u$ to $z$ and from $z$ to $w$
            for each odd integer $d$ and vertex $v \notin A$ which is a predecessor of $z$
            along a path of length $d$   do
              for all $q$ such that 'reduce$q$' $\in$ACTION(STATE($v$),$a_{i+1}$)  do
                for all vertices $t$ such that there exists a directed walk of
                length $2|RHS(q)| - d$ from $v$ to $t$ that goes through vertex $z$
                do add $(t, q)$ to $R$
         end
     end
 else /* i.e. when there does not exist $u \in U_i$ such that STATE($u$)= $s$ */
     begin
        create in $\Gamma$ two vertices $u$ and $z$ labelled $s$ and $N$ respectively
        create two arcs in $\Gamma$ from $u$ to $z$ and from $z$ to $w$
        add $u$ to both $A$ and $U_i$
     end

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

6 GLR parsing with a modified DFA

In this section we describe a general parsing algorithm which we prove is correct on all context free grammars and input strings. The algorithm is based on Tomita's Algorithm 1, incorporating the minor additions given in Algorithm 1e above, but reductions whose final symbols are nullable are applied when the last non-nullable symbol is seen. I.e. if we have a production rule $A ::= \alpha BC$ where $B\overset{*}{\Rightarrow}\epsilon$ and $C\overset{*}{\Rightarrow}\epsilon$ we treat the item $(A ::= \alpha \cdot BC, b)$ as though it were a reduction.

6.1 The reduction modified DFA and an example

We construct the LR(1) DFA for a grammar in the normal fashion. However, in the corresponding table we store the reductions slightly differently. As usual, if there is a transition from state h to state k labelled x then we store sk or gk in position (h, x) of the table. If h contains an item of the form $(A ::= x_1 \ldots x_m \cdot B_1 \ldots B_t,\ b)$, where $t = 0$ or $B_p\overset{*}{\Rightarrow}\epsilon$ for all $1 \leq p \leq t$ and $A ::= x_1 \ldots x_m B_1 \ldots B_t$ is production rule number j, then we store the action (rj, m) in position (h, a) of the table.

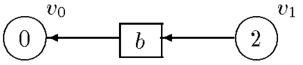
We illustrate this using the following example from Section 4.4

$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & bA \\ A & ::= & aAB \mid \epsilon \\ B & ::= & \epsilon \end{array}$$

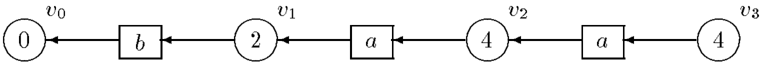
	$	a	b	A	B	S
0			s2			g1
1	acc					
2	(r3,0)/(r1,1)	s4		g3		
3	(r1,2)					
4	(r3,0)/(r2,1)	s4		g5		
5	(r4,0)/(r2,2)				g6	
6	(r2,3)					

We describe the construction of the GSS from this table with input string baa.

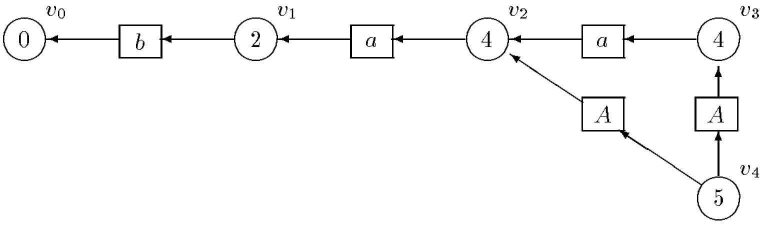
We begin with the base node, v_0, labelled 0. The action in position $(0, b)$ is $s2$ so we create a new state node, v_1, labelled 2 and a new symbol node labelled b.



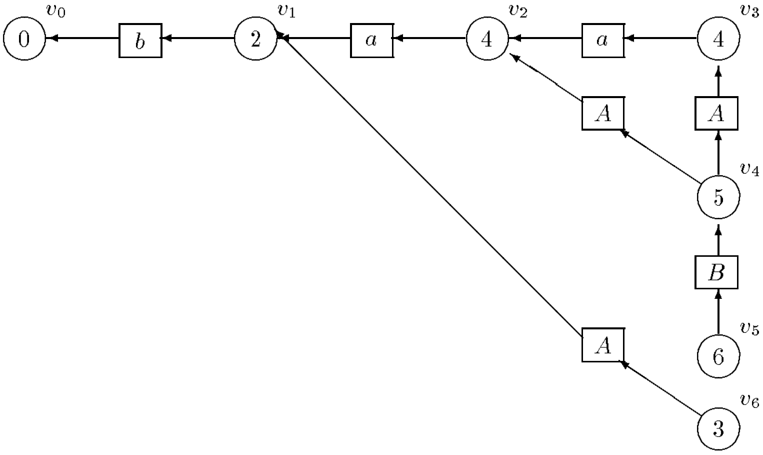
We then read the next two input symbols, aa, and create new symbol nodes v_2 and v_3 in a similar fashion.



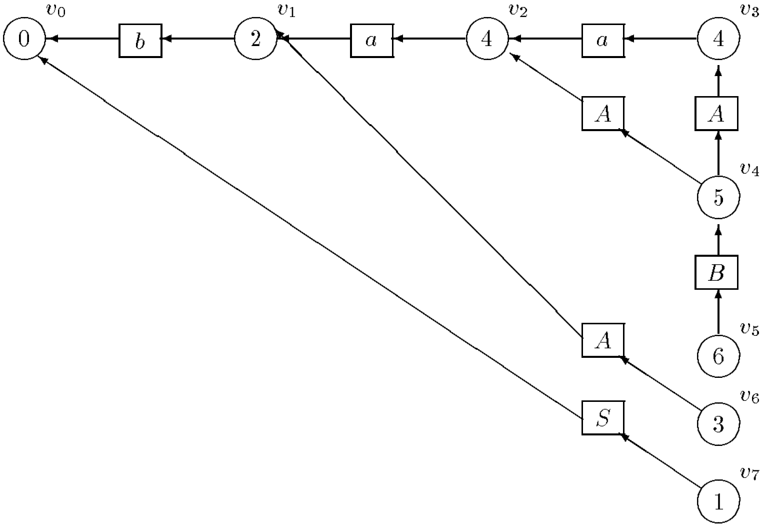
When the node v_3 is created the entry $(r3, 0)$ in position $(4, \$)$ results in the construction of a new node, v_4, and $(r2, 1)$ adds another path from v_4.



The reduction $(r4, 0)$ results in a new node v_5. The reduction $(r2, 2)$ does not need to be applied down the path from v_4 through v_3 because this action was already taken when the reduction $(r2, 1)$ was applied. (If we do apply $(r2, 2)$ down this path we will simply find that the path of length 2 from v_4 to v_2 already exists.) Thus the reduction $(r2, 2)$ is just applied down the path through v_2, resulting in a new node v_6.



In the same way, the reduction $(r2, 3)$ in position $(6, \$)$ does not need to be applied down the path from v_5 through v_4. The reduction $(r1, 2)$ is applied from node v_6 to generate node v_7, and the construction is complete.



Since v_7 is the accept state of the DFA, the string is accepted.

6.2 Generalised reduction modified LR parser (GRMLR)

```
input: reduction-modified DFA, input string $a_1 \dots a_n\$$

PARSER {
  create a state node $v_0$ labelled with the start state 0 of the DFA.
  set $U_0 = \{v_0\}$, $R = \emptyset$, $Q = \emptyset$, $a_{n+1} = \$$, $U_1 = \emptyset$, ..., $U_n = \emptyset$
  if $sk \in$table$(0, a_1)$ add $(v_0, k)$ to $Q$
  for all $(rj, 0) \in$table$(0, a_1)$ add $(v_0, X, 0)$ to $R$, where $X$ is the LHS of $j$
  for $i = 0$ to $n$ while $U_i \neq \emptyset$ do {
    $A = U_i$
    while $R \neq \emptyset$ do REDUCER($i$)
    do SHIFTER(i)
  }
  if the DFA accepting state is in $U_n$ report success else report failure
}

REDUCER($i$) {
  remove $(v, X, m)$ from $R$
  find the set $\chi$ of state nodes which can be reached from $v$ along a path of
       length $2(m-1)$, or length 0 if $m = 0$
  for each state node $u \in \chi$ do {
    let $k$ be the label of $u$ and let $gl$ be the entry in table$(k, X)$
    if there is a node $w \in U_i$ with label $l$ {
      if there is not a path of length 2 from $w$ to $u$ {
        if there is a path of length 2 from $w$ to a node in $U_j$
          make the symbol node in the middle of this path a predecessor of $u$
        else create a symbol node labelled $X$ which is a successor of $w$
                    and a predecessor of $u$
        if $m \neq 0$ { for all $(rj, t)$ in table$(l, a_{i+1})$ where $t \neq 0$
                        and $B$ is the LHS of $j$, add $(u, B, t)$ to $R$ }
      }
      else {
        create a new state node, $w$, in the GSS labelled $l$ and a new symbol
                                        node, $y$, labelled $X$
        make $y$ a successor of $w$ and a predecessor of $u$
        add $w$ to $U_i$
        if $sh \in$table$(l, a_{i+1})$ add $(w, h)$ to $Q$
        for all reductions $(rj, 0)$ in table$(l, a_{i+1})$ add $(w, B, 0)$ to $R$ where
                                                    $B$ is the LHS of $j$
        if $m \neq 0$ { for all $(rj, t)$ in table$(l, a_{i+1})$ where $t \neq 0$
                       and $B$ is the LHS of $j$, add $(u, B, t)$ to $R$ }
      }
  }
}

SHIFTER($i$) {
  if $i \neq n$ {
    $Q' = \emptyset$ (a temporary set to hold new shifts)
    while $Q \neq \emptyset$ do {
      remove an element $(v, k)$ from $Q$
      if there is $w \in U_{i+1}$ with label $k$ {
          let $u$ be the symbol node which is the successor of $w$
          make $u$ a predecessor of $v$
          for all $(rj, t)$ in table$(k, a_{i+2})$ where $t \neq 0$ add
                $(v, B, t)$ to $R$, where $B$ is the LHS of $j$
      }
      else {
          create a new state node, $w$, in the GSS labelled $l$ and a new symbol
                                    node, $u$, labelled $a_{i+1}$
          make $u$ a successor of $w$ and a predecessor of $v$
          add $w$ to $U_i$
          if $sh \in$table$(k, a_{i+2})$ add $(w, h)$ to $Q'$
          for all $(rj, t)$ in table$(k, a_{i+2})$ where $t \neq 0$
                      add $(v, B, t)$ to $R$, where $B$ is the LHS of $j$
          for all $(rj, 0)$ in table$(k, a_{i+2})$ add
                      $(w, B, 0)$ to $R$, where $B$ is the LHS of $j$
      }
    }
    copy $Q'$ into $Q$
  }
}
```

Our algorithm is in the same style as Tomita's Algorithm 1 except we do not have a separate ACTOR which processes nodes in the GSS. In our algorithm, when a node v is created, any shift which is possible on the next input symbol is immediately recorded in the set Q, for execution once the input related reduction-closure has been completed.

Reductions of length m from v must be applied down all paths from v of length $2m$. If $m = 0$ there can only ever be one such path, the empty path. If $m \geq 1$ then new paths of length $2m$ from v are created every time a new successor node is added to v. Thus when a node is created all reductions of length 0 are recorded in R. If the node has been created as the result of the application of a reduction of length 0 (a reduction corresponding to a rule $B ::= \cdot\alpha$) then any reductions of length greater than 0 will already have been applied from a previous node (this is the role of the new reduction items $(A ::= \alpha \cdot B, b)$ etc in the DFA table) so no reductions of length greater than 0 are recorded.

If the node has been constructed via a reduction of length greater than 0 then we record in R all reductions of length greater than 0, together with the second node along the path from v, for subsequent execution.

If a new path from an existing node v is created then this must be as a result of applying a reduction. If this reduction is of length 0 then any reductions down the new path will have already been recorded. If the reduction is of length greater than 0 then the reductions of length greater than 0 in v and this new path must be recorded in R for subsequent execution. (The reductions of length 0 are not recorded in this case because adding a new edge from v does not create a new path of length 0).

In order to generate more efficient derivation trees, we have also written REDUCER(i) so that in the case where we have two paths of length 2 from the same node in U_i to different nodes in U_j then the first half of these two paths are merged. This results, in certain cases, in fewer symbol nodes in our GSS than in the corresponding GSS produced with Algorithm 1e. Of course, there is also an additional cost in that paths of length 2 from a given node have to be searched to check whether their end nodes lie in the appropriate U_j. In order to properly compare our algorithm with Algorithm 1e, we have also implemented another version of our REDUCER(i) which does not merge non-terminal symbol nodes in this way. This version is given in Section 6.3.

6.3 The efficiency of the algorithm

On first inspection we might expect our new algorithm to be less efficient than Tomita's because, in general, there will be significantly more conflicts in the underlying parse table. This could lead to more possibilities which need to be investigated and hence to a larger GSS. However, we claim that the GSS constructed using our method is identical to that constructed using Algorithm 1e in the cases where the latter algorithm works correctly.

Although, for right nullable grammars, there are more conflicts in the reduction modified table, these conflicts result only from moving the point at which a reduction is applied. These reductions would have been applied in the original case after some 'shifting' of ϵ-matching non-terminals on to the stack, and without consuming any further input symbols. Thus these conflicts do not cause any additional reductions to be applied over and above what would have eventually been done anyway. This point is illustrated further when we discuss the relationship between reduction modified tables and LR(1) grammars in Section 6.5, and is essentially a consequence of our proof of the correctness of our algorithm given in Section 6.4 below.

Furthermore, the GSS construction process is more efficient in our case in the sense that the amount of graph traversal needed to construct the GSS is less when the grammar contains right nullable rules. This is because reductions via rules of the form $A ::= \alpha\beta$ where $\beta \overset{*}{\Rightarrow} \epsilon$ are applied down paths of length $|\alpha|$ rather than paths of length $|\alpha\beta|$.

In Section 7 we shall give examples from real grammars such as a C grammar and compare the effects of our algorithm with Algorithm 1e. In order to compare like with like, we shall use a slightly different version of our algorithm in which, as in Algorithm 1e, the symbol nodes corresponding to non-terminals are not merged.

```
REDUCER($i$) {
  remove $(v, X, m)$ from $R$
  find the set $\chi$ of state nodes which can be reached from $v$ along a path of
      length $2m$
  for each state node $u \in \chi$ do {
    let $k$ be the label of $u$ and let $gl$ be the entry in table$(k, X)$
    if there is a node $w \in U_i$ with label $l$ {
      if there is not a path of length 2 from $w$ to $u$ {
        create a symbol node labelled $X$ which is a successor of $w$
                        and a predecessor of $u$
        if $m \neq 0$ { for all $(rj, t)$ in table$(l, a_{i+1})$ where $t \neq 0$
                        and $B$ is the LHS of $j$, add $(u, B, t)$ to $R$ }
      }
    else {
        create a new state node, $w$, in the GSS labelled $l$ and a new symbol
                                node, $y$, labelled $X$
        make $y$ a successor of $w$ and a predecessor of $u$
        add $w$ to $U_i$
        if $sh \in$table$(l, a_{i+1})$ add $(w, h)$ to $Q$
        for all reductions $(rj, 0)$ in table$(l, a_{i+1})$ add $(w, B, 0)$ to $R$ where
                                        $B$ is the LHS of $j$
        if $m \neq 0$ { for all $(rj, t)$ in table$(l, a_{i+1})$ where $t \neq 0$
                        and $B$ is the LHS of $j$, add $(u, B, t)$ to $R$ }
      }
   }
}
```

The experiments demonstrate that the GSS constructed by our algorithm with the above version of REDUCER(i) is the same size as that constructed by Tomita's algorithm and that the graph searching required to construct the GSS in this case is less than is required for Algorithm 1e, for right nullable grammars.

6.4 Correctness of the algorithm

A language recognition algorithm is correct for a given context free grammar if, given any input string, the algorithm terminates and reports success if the input string is in the language generated by the grammar, and terminates and reports failure otherwise.

Our proof of the correctness of our algorithm depends on the correctness of the standard stack and table based parsing technique. We shall give a formal definition of what we mean by the language accepted by a (possibly non-deterministic) table based parser, and we shall prove that, for all context free languages, our algorithm (deterministically) accepts exactly the language accepted by the table based parser.

We begin with a general definition of a parse table. We then define the operation of a stack based machine with such a table on a given input string, and then we define the language accepted by this machine. These definitions are just extensions of the standard definitions for LR(1) tables to include tables which may have conflicts and reductions applied on partially recognised handles.

6.4.1 RM-Parse tables

A *parse table*, for a grammar Γ whose rules are uniquely numbered, is a table whose rows are labelled with a strictly increasing finite sequence of integers starting from 0, and whose columns are labelled with the terminals and non-terminals of the grammar together with the special end of file symbol \$. The entries in the table are sets of *actions*. These actions are of the form sk, gk, (rj, m), and acc, where k is a row number, j is the number of a grammar rule and m is an integer which lies between 0 and the length of the right hand side of j. Entries in columns labelled with non-terminals can contain at most one element, which must be of the form gk. Entries in columns labelled with terminals or \$ can contain up to one action of the form sk and arbitrarily many actions of the form (rj, m). In the column labelled \$ the entries can also contain the action acc.

We call the parse table constructed from a grammar Γ using LR(1)-items in the standard way the *LR(1)-table* for Γ. For non-LR(1) grammars the LR(1)-table will contain some entries with more than one action (usually referred to as *conflicts*). In the LR(1) parse table all actions of the form (rj, m) will have m equal to the length of the right hand side of j, so these are usually written just as rj.

As described in Section 6.1, we define the *reduction modified LR(1)-table* (or the *RM-table*) to be the table obtained by taking the LR(1) table and adding extra actions of the form (rj, k) to the entry in position (h, a) of the table if the DFA state h contains an LR(1)-item of the form $(A ::= \alpha \cdot \beta, a)$, where $\beta \overset{*}{\Rightarrow} \epsilon$, α has length k, and $A ::= \alpha\beta$ is rule number j in the grammar.

6.4.2 Table based parsers

We now define a (non-deterministic) machine which takes a parse table and an input string and traverses the table according to the actions in the table entries. This is just a straightforward extension of the standard LR(1) stack based parser to include tables with multiple entries and reductions of length less than the length of the right hand side of the reduction rule.

We define a *table based parser* to be a stack based machine with an associated parse table which takes as input a string of symbols. Initially the stack contains the label, 0, of the first row of the table.

At any step in an execution of the parser the stack will contain an alternating sequence of row labels and column labels from the table, with 0 at the bottom and row label at the top. An execution step consists of looking at the current symbol, a say, in the input string and the integer, h say, on the top of the stack, (non-deterministically) selecting an action from the set in position (h, a) of the table, and carrying this action out.

If there are no actions in the set then the parser terminates and reports a failure. If the action is acc then the parser terminates and reports acc. If the action is sk then the parser pushes a and then k on to the stack and advances the input pointer. If the action is (rj, m), where A is the symbol on the left hand side of rule j, then the parser pops $2m$ symbols off the stack, reads the new top-of-stack symbol, l say, and then reads the entry in position (l, A) of the table.

If there is no action in this entry then the parser terminates and reports failure. If the action is gk then then parser pushes A and then k on to the stack.

An *execution path* of a table based parser for a given input string is a sequence of execution steps which start with 0 on the stack and the input pointer at the beginning of the string. A string u is *accepted* by a table based parser if, on input $u\$$, there is some execution path of the parser which results in the action *acc*.

An LR(1)-parser for a grammar Γ is a table based parser whose associated table is the LR(1)-table for Γ. An RM-parser for Γ is a table based parser whose associated table is the RM-table for Γ. We shall now show that the LR(1)- and RM-parsers for a given context free grammar are equivalent in the sense that they accept the same set of strings.

6.4.3 Equivalence of LR(1)- and RM-parsers

In order to prove that the LR(1)- and RM-parsers accept the same set of strings in the case where the associated tables have been generated from the same grammar, we need the following lemma, which addresses the behaviour of the LR(1)-parser on right nullable rules.

Lemma 1 *Suppose that there is a execution path of an LR(1)-parser which results in a stack of the form* $0, x_1, h_1, x_2, h_2, \dots x_q, h_q$ *and the input pointer pointing at symbol* a*. Suppose also that the state* h_q *contains an LR(1)-item of the form* $(A ::= \alpha \cdot \beta, a)$*, where* $\beta \overset{*}{\Rightarrow} \epsilon$*. Then there are continuations of the execution path*

1. *in which the input pointer is not moved and the stack takes the form*

$$0, x_1, h_1, \dots, x_q, h_q, y_1, k_1, \dots, y_p, k_p$$

   where $\beta = y_1 \dots y_p$*, and*

2. *in which the input pointer is not moved and the stack takes the form*

$$0, x_1, h_1, \dots, x_i, h_i, A, h$$

   where $|\alpha| = (q - i)$ *and the action in position* (h_i, A) *of the LR(1)-table is* gh*.*

Proof We prove part (1) by induction on the length of the derivation $\beta \overset{n}{\Rightarrow} \epsilon$, and then we observe that part (2) follows directly from part (1).

Suppose first that $n = 0$, so $\beta = \epsilon$. Then the new stack is the same as the original one, and part (1) is trivially true.

Now suppose that $n \geq 1$ and that the result is true for items of the form $(X ::= \gamma \cdot \delta, b)$ where $\delta \overset{d}{\Rightarrow} \epsilon$ and $d < n$. Since $n \geq 1$ we must have $\beta = y_1 \beta'$ where $y_1 \neq \epsilon$ and $y_1 \Rightarrow \tau \overset{d}{\Rightarrow} \epsilon$, where $d < n$. From the standard construction of the LR(1)-DFA states, we have that h_q must contain the item $(y_1 ::= \cdot \tau, a)$, and so, by induction, we can extend the execution path so that the stack is of the form

$$0, x_1, h_1, \dots, x_q, h_q, z_1, k'_1, \dots, z_r, k'_r$$

where $\tau = z_1 \dots z_r$, without moving the input pointer. Also from the LR(1)-DFA construction process we must have that k'_r contains the item $(y_1 ::= \tau\cdot, a)$, hence entry in position (k'_r, a) of the table will contain a reduction by this rule, and so there is an execution step in which the input pointer is not moved and the stack becomes

$$0, x_1, h_1, \dots, x_q, h_q, y_1, k_1$$

where gk_1 is the entry in position (h_q, y_1) of the LR(1)-table. From the LR(1) construction method we have that k_1 contains the item $(A ::= \alpha y_1 \cdot \beta', a)$, where $\beta' = y_2 \dots y_p$. We have $\beta' \overset{f}{\Rightarrow} \epsilon$ where $1 + d + f = n$, so $f < n$ and, by induction, we can extend the execution path so that the stack is of the form

$$0, x_1, h_1, \dots, x_q, h_q, y_1, k_1, \dots, y_p, k_p$$

without moving the input pointer. This proves part (1).

Now, by part (1), without moving the input pointer we can extend the execution path so that the stack is of the form in part (1). Since each y_t is a non-terminal and h_q contains the item $(A ::= \alpha \cdot y_1 \dots y_p, a)$, the LR(1)-DFA state construction process guarantees that h_t will contain the item $(A ::= \alpha y_1 \dots y_t \cdot y_{t+1} \dots y_p, a)$. Thus position (k_p, a) of the LR(1)-table will contain the action to reduce by the rule $A ::= \alpha\beta$. Thus there is an execution in which the input pointer is not moved, $2(k - i + p)$ symbols are popped off the stack, and then A and the state h, such that gh is contained in position (h_i, a) of the LR(1)-table, are pushed on to the stack. This gives the stack

$$0, x_1, h_1, \dots, x_i, h_i, A, h$$

as required for part (2).

Theorem 1 *For any context free grammar Γ, the LR(1)- and RM-parsers for Γ are equivalent in the sense that they accept the same set of strings.*

Proof Since the LR(1)-table for Γ is a subset of the RM-table for Γ (all of the actions in the LR(1)-table are also in the RM-table), for any execution path through the LR(1)-parser there is an identical execution path through the RM-parser. Thus any string accepted by the LR(1)-parser for Γ will also be accepted by the RM-table based parser for Γ.

To show that any string which is accepted by the RM-parser for Γ is also accepted by the LR(1)-parser, we show that if, on input $a_1 \dots a_{i+1}$ there is an execution path through the RM-parser which results in the stack

$$0, x_1, h_1, \dots, x_p, h_p$$

and input pointer pointing at a_{i+1} then there is an execution path through the LR(1)-table based parser which results in the same stack and pointer position.

If the execution path is empty then both parsers have stack containing just the state 0 and the input pointer pointing at the first input symbol, so the result is true.

Now suppose that the execution path in question consists of $q \geq 1$ execution steps and that the result is true for paths with less than q steps. Suppose that the first $(q-1)$ execution steps in the path resulted in the stack

$$0, y_1, k_1, \ldots, y_t, k_t$$

with the input pointer pointing at b, and that at the last execution step the action *act* was selected. If *act* is of the form *sk* then to arrive at the given stack and pointer position, we must have $b = a_i$ and $y_l = x_l$, $k_l = h_l$ for $1 \leq l \leq t = p-1$. Otherwise, we must have $b = a_{i+1}$.

By induction we can assume that there is an execution path, on input $a_1 \ldots a_n\$$, in the LR(1)-table based parser which results in the stack

$$0, y_1, k_1, \ldots, y_t, k_t$$

and the input pointer pointing at b.

If *act* was *acc* then the input pointer must have been pointing at a_{i+1}, we must have $\$ = a_{i+1}$ and, since *acc* does not alter the stack, $y_l = x_l$, $k_l = h_l$ for $1 \leq l \leq t = p$. From the construction of the RM-table we must also have that $k_t = h_p$ is the accepting state of the underlying LR(1)-DFA. Thus position $(h_p, \$)$ of the LR(1)-table will also contain the action *acc*, and the required execution path exists in the LR(1)-table based parser.

If *acc* was *sk* then, as we have already noted, the previous step would have been carried out on the stack

$$0, x_1, x_1, \ldots, x_{p-1}, h_{p-1}$$

with the input pointer pointing at a_i. The action sh_p would also be in position (h_{p-1}, a_i) of the LR(1)-table, so the LR(1)-parser could also extend its execution path in a way that results in stack

$$0, x_1, x_1, \ldots, x_p, h_p$$

and input pointer pointing at a_{i+1}, as required.

Finally suppose that *act* was (rj, m), so that at the previous step in the execution path the stack had the form

$$0, x_1, h_1, \ldots, x_{p-1}, h_{p-1}, z_1, g_1, \ldots, z_m, g_m$$

and the input pointer was pointing at a_{i+1}. By induction, there is an execution of the LR(1)-parser on the same input which results in the same stack and input pointer position. Since the next step in the execution path of the RM-parser results in the stack

$$0, x_1, h_1, \ldots, x_p, h_p$$

we must have that x_p is the left hand side of rule j and that position (h_{p-1}, a_{i+1}) of the RM-table, and hence of the LR(1)-table, contains the action gh_p. Furthermore, by construction of the RM-table, the DFA state g_m must contain an item $(x_p ::= \alpha \cdot \beta, a_{i+1})$, where $|\alpha| = m$ and $\beta \overset{*}{\Rightarrow} \epsilon$. Then, by Lemma 1, there is a continuation of the LR(1)-parser's execution path with results in the stack

$$0, x_1, h_1, \ldots, x_p, h_p$$

and input pointer remaining at a_{i+1}, as required.

6.4.4 Correctness of the GRMLR algorithm

We have already shown that an RM-parser accepts the same language as the LR(1)-parser for the same grammar, and we assume that the latter accepts precisely the language generated by the underlying grammar. The problem is that both of these machines are, in general, non-deterministic; for a given input string we cannot tell whether there is an execution path of the machine which results in *acc*.

We define a *determining algorithm* for a table based parser to be an algorithm which determines whether or not, given a grammar and an input string, there is an execution path through the table based parser which results in *acc*.

Tomita's Algorithm 2 was designed to be an LR(1)-table determining algorithm, but it fails to terminate on grammars with hidden left recursion. Algorithm 1e was also designed to be an LR(1)-table determining algorithm, but the problem here is that with certain grammars which contain right nullable rules Algorithm 1e erroneously reports that no execution path exists.

We define a determining algorithm to be *correct* for a table based parser if, given any input string, it terminates and reports success if there is an execution path of the table based parser which reports *acc*, and terminates and reports failure otherwise. It is believed that Tomita's Algorithm 2 is correct for LR(1)-parsers whose tables have been generated from grammars without hidden left recursion, and that Algorithm 1e is correct for LR(1)-parsers whose tables have been generated from grammars without right nullable rules. We shall prove that the GRMLR algorithm given in Section 6.2 is a correct determining algorithm for RM-parsers whose associated tables have been generated from any context free grammar.

The problem with Tomita's Algorithm 2 is that it fails to terminate in certain cases. It is also true that Aycock and Horspool's algorithm fails to terminate on grammars which contain hidden left recursion [AH99]. We begin by proving that the GRMLR algorithm terminates for all RM-parsers and all input strings.

Lemma 2 *For all context free grammars, the GRMLR algorithm given in Section 6.2 terminates for all input strings.*

Proof We suppose that the algorithm is using an RM-table with N rows (states), constructed from a context free grammar Γ, and that the input string is $a_1 \ldots a_n$.

First we calculate an upper bound on the size of the GSS.

Each set U_i of level i nodes has only one node labelled with each state number, so it contains at most N nodes. Thus the GSS contains at most $(n+1)N$ state nodes. All edges from a node $u \in U_i$ are the first edge in a path of length 2 to a node in some U_j where $j \leq i$. At most one edge is added from a node $u \in U_i$ by the SHIFTER, all other edges are added by the REDUCER and this checks for the existence of a path of length 2 before adding the edge. Thus there is at most one path of length 2 from U_i to each of the $(i+1)N$ nodes in the U_j with $j \leq i$. Thus there are at most $(i+1)N$ edges from each node in U_i, and hence there are at most $(i+1)N^2$ symbol nodes which are successors of nodes in U_i. So the GSS contains at most

$$(\sum_{i=0}^{n}(i+1)N^2) \quad + \quad (n+1)N \quad = \quad \frac{(n+1)((n+2)N^2+2N)}{2}$$

nodes (state and symbol nodes).

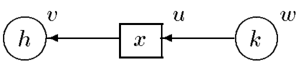
The GSS is a bipartite graph and every edge has a symbol node either as its source node or its target node. The GSS construction guarantees that symbol nodes have only one predecessor, so there are as many edges whose target is a symbol node as there are symbol nodes. All the edges from a given symbol node are guaranteed to have target nodes in the same U_j, thus each symbol node is the source node of at most N edges. Thus the GSS contains at most the following number of edges.

$$(N+1)(\sum_{i=0}^{n}(i+1)N^2) \quad = \quad \frac{(n+1)(n+2)N^2(N+1)}{2}.$$

The **for** loops in REDUCER(i) iterate over finite sets which are not modified during the execution of the loop, thus this function will always terminate. The **for** loops in SHIFTER(i) iterate over table entries, and these are fixed. The **while** loop in SHIFFTER(i) removes an element from Q at each iteration, and doesn't add any elements to Q, thus SHIFTER(i) always terminates. So to show that the algorithm always terminates we need to show that the **while** loop in the function PARSER terminates for each value of i. Each time REDUCER(i) executes it removes an element from the set R of pending reductions. Looking at the structure of REDUCER(i), we see that it only adds elements to R when a new edge is created in the GSS. We have already seen that there can only be a finite number of edges in the GSS, so REDUCER(i) must eventually stop adding elements to R, but continue removing one each time it is executed. Thus, eventually, we will have $R = \emptyset$ and the **while** loop will terminate, as required.

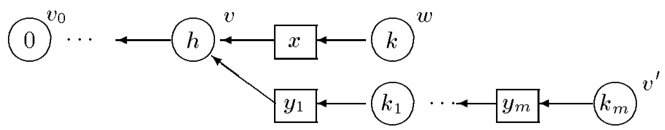
In order to show that the RM-table determining algorithm given in Section 6 accepts precisely the language accepted by the RM-table (and hence by Lemma1 the LR(1)-table) based parser, we need the following lemma which addresses the impact of nullable non-terminals on the GSS and on the parse table.

Lemma 3 *If there is a path*



in the GSS constructed with an RM-table and input $a_1 \ldots a_n$, in which w and v lie in the same U_i, then $x \overset{*}{\Rightarrow} \epsilon$. Furthermore, the state h in the LR(1) DFA contains an item $(x ::= \cdot\gamma, a_{i+1})$, where $\gamma \overset{*}{\Rightarrow} \epsilon$. Hence the RM-table contains the element $(rf, 0)$ in position (h, a_{i+1}), where f is the rule $x ::= \gamma$, and position (h, x) contains the element gk.

Proof Since $w, v \in U_i$, the edge (u, v) must have been constructed as a result of processing a reduction (v', x, m), where there is a path of length $2(m-1)$ (or of length 0 if $m = 0$) from v' to u,

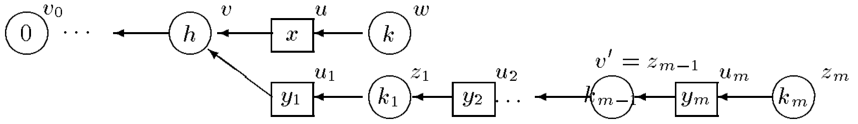


and we must have that position (h, x) of the RM-table contains the element gk.

Looking at REDUCER(i) and SHIFTER(i) we see that for (v', x, m) to be in R at this point we must have either that

1. $(rf, 0)$ lies in position (k_m, a_{i+1}) of the RM-table, where f is a rule of the form $x ::= \gamma$, so $m = 0$, $v' = v$, $k_m = h$, and $\gamma \overset{*}{\Rightarrow} \epsilon$, or

2. there is a node $z_m \in U_i$ with label k_m, such that there is a path of length 2 from z_m to v' and (rf, m) lies in position (k_m, a_{i+1}) of the RM-table, where f is a rule of the form $x ::= y_1 \ldots y_m \beta$, and $\beta \overset{*}{\Rightarrow} \epsilon$.



Each of the nodes z_d must lie in U_j for some $j \leq i$. Since $v \in U_i$ and there is a path from each of these nodes to v, in fact they must all lie in U_i.

We now prove the results by induction on the order in which the edges in the GSS were created.

If (w, u) and (u, v) are the first edges created then $v = v_0$ and $w \in U_0$. Since there are no paths in the GSS before this one is created, we must be in Case 1 above, $x \Rightarrow \gamma \overset{*}{\Rightarrow} \epsilon$, and $(rf, 0)$ lies in position $(0, a_1)$. Thus, by the RM-table construction rules, we must have $(x ::= \cdot\gamma, a_1)$ in state 0.

Now suppose that the result is true for edges which were created before (u, v).

If (v', x, m) falls in to Case 1 above, then we have $x \Rightarrow \gamma \overset{*}{\Rightarrow} \epsilon$ and $(rf, 0)$ lies in position $(0, a_{i+1})$. So we must have $(x ::= \cdot\gamma, a_{i+1})$ in state h, as required.

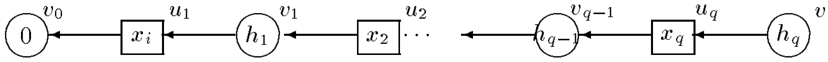
If (v', x, m) falls in to Case 2 then all of the edges (u_d, z_{d-1}), where $1 \leq d \leq m$ and $z_0 = v$, were created before the edge (u, v), and so, by induction, $y_d \overset{*}{\Rightarrow} \epsilon$ for $1 \leq d \leq m$. Thus we have $x \Rightarrow y_1 \ldots y_m \beta \overset{*}{\Rightarrow} \epsilon$, and since (rf, m) lies in position (k_m, a_{i+1}) of the RM-table, k_m contains the item $(x ::= y_1 \ldots y_m \cdot \beta, a_{i+1})$. Then, by the correspondence between paths of length 2 in the GSS and transitions in the LR(1) DFA, we have that k_{d-1} contains the item $(x ::= y_1 \ldots y_{d-1} \cdot y_d \ldots y_m y \beta, a_{i+1})$. Thus $h = k_0$ contains the item $(x ::= \cdot y_1 \ldots y_m y \beta, a_{i+1})$ and $y_1 \ldots y_m y \beta \overset{*}{\Rightarrow} \epsilon$, as required.

Theorem 2 *Given any context free grammar* Γ *and any input string, u, the GRMLR algorithm given in Section 6.2 terminates and reports success if there is an execution of the LR(1)-parser for* Γ *which results in acc, and terminates and reports failure otherwise.*

Proof From Theorem 1 it is sufficient to show that the GRMLR algorithm terminates and reports success if there is an execution of the RM-table based parser with results in *acc*, and terminates and reports failure otherwise. We have already shown that the algorithm always terminates, so we need to show that it reports success on input u if and only if the there is an execution path through the RM-parser which results in *acc*.

Let G be the GSS constructed from the RM-table for Γ and input $a_1 \dots a_n$, let $a_{n+1} = \$$, and let v_0 be the base node of G, the first node constructed.

($\Rightarrow$): We suppose that the GRMLR algorithm reports success. We shall show that, if there is a node $v \in U_i$ and a path



in G then there is an execution path through the RM-parser on input $a_1 \dots a_{n+1}$ which results in the stack

$$0, x_1, h_1, \dots, x_q, h_q$$

and input pointer pointing at a_{i+1}. Then, since the GSS reports success, there is a node $v_n \in U_n$ whose label, h_n, is the accept state of the RM-table. Thus there is an execution path through the RM-parser which results in h_n on the top of the stack and the input pointer pointing at $a_{n+1} = \$$. Since *acc* lies in position $(h_n, \$)$ of the RM-table, this execution path can be extended to result in *acc*.

The proof is by induction on the order in which the edges in G are created.

If the path from v under consideration has no edges then we must have $v = v_0$ and $i = 0$. The start configuration of an RM-table based parser ensures that there is an (empty) execution path which results in 0 on the stack and the input pointer pointing to a_1, so the result is trivially true.

Now suppose that the edge (u_j, v_{j-1}) was the last of the edges in the path to be created and suppose that the result is true for all nodes and paths which contain only edges created before (u_j, v_{j-1}).

If the edge (u_j, v_{j-1}) was created by the SHIFTER then, since it was the last edge in the path to be created, we must have $j = q$, $v_{q-1} \in U_{i-1}$ and sh_q must be an entry in position (h_{q-1}, a_i) of the RM-table. Then, since all the edges on the path from v_{q-1} to v_0 were created before the edge (u_q, v_{q-1}), by induction there is an execution path through the RM-parser on input $a_1 \dots a_{n+1}$ which results in the stack

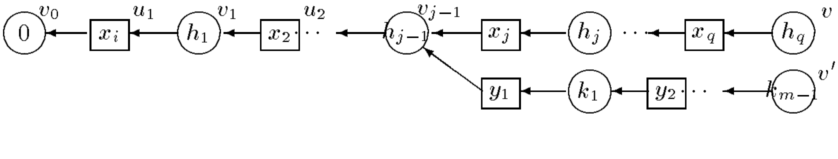
$$0, x_1, h_1, \dots, x_{q-1}, h_{q-1}$$

and input pointer pointing at a_i. Since the action sh_q lies in position (h_{q-1}, a_i) of the RM-table, this execution path can then be extended to give the stack

$$0, x_1, h_1, \dots, x_{q-1}, h_{q-1}, x_q, h_q$$

leaving the input pointer pointing at a_{i+1}.

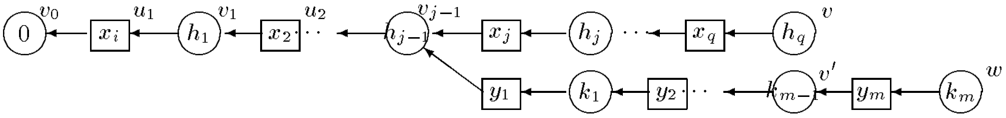
Now suppose that the edge (u_j, v_{j-1}) was created by the REDUCER. Since this is assumed to be the last edge created and since $v_q \in U_i$, this edge must have been created by REDUCER(i) while processing an element (v', x_j, m) and there is a path



in G. Looking at REDUCER(i) and SHIFTER(i) we see that for (v', x_j, m) to be in R at this point we must have either that

1. $(rf, 0)$ lies in position (k_m, a_{i+1}) of the RM-table, where f is a rule $x_j ::= \gamma$, so $m = 0$ and $v' = v_{j-1}$, or

2. there is a node $w \in U_i$ with label k_m, such that there is a path of length 2 from w to v' and (rf, m) lies in position (k_m, a_{i+1}) of the RM-table, where f is a rule of the form $x_j ::= \alpha\gamma$ and $|\alpha| = m$.



In Case 1, by induction, there is an execution path through the RM-parser on input $a_1 \ldots a_{n+1}$ which results in the stack

$$0, x_1, h_1, \ldots, x_{j-1}, h_{j-1}$$

without moving the input pointer. Since $(rf, 0)$ lies in position (h_{j-1}, a_{i+1}) of the RM-table, there is a continuation of the execution path which results in the stack

$$0, x_1, h_1, \ldots, x_{j-1}, h_{j-1}, x_j, h_j$$

without moving the input pointer. In Case 2, since all of the path from w to v_{j-1} was created before the edge (u_j, v_{j-1}), by induction, there is an execution path through the RM-parser on input $a_1 \ldots a_{n+1}$ which results in the stack

$$0, x_1, h_1, \ldots, x_{j-1}, h_{j-1}, y_1, k_1, \ldots, y_m, k_m$$

and input pointer pointing at a_{i+1}. Since $(rf, m+1)$ lies in position (l, a_{i+1}) of the RM-table, there is a continuation of the execution path which again results in the stack

$$0, x_1, h_1, \ldots, x_{j-1}, h_{j-1}, x_j, h_j$$

without moving the input pointer.

From the operations in REDUCER(i) we see that we must have $v_j \in U_i$, and hence $v_d \in U_i$ for $j < d \leq q$. Then, by Lemma 3, the RM-table contains entries $(rf_d, 0)$ in position (h_{d-1}, a_{i+1}), where f_d has left hand side x_d, and position (h_{d-1}, x_d) contains the element gh_d. Thus we can continue the above execution path to generate stacks

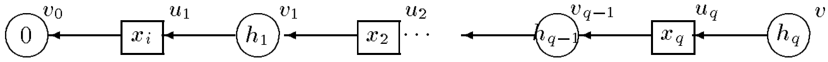
$$\begin{gathered} 0, x_1, h_1, \ldots, x_{j-1}, h_{j-1}, x_j, h_j, x_{j+1}, h_{j+1} \\ 0, x_1, h_1, \ldots, x_{j+1}, h_{j+1}, x_{j+2}, h_{j+2} \\ \vdots \\ 0, x_1, h_1, \ldots, x_q, h_q \end{gathered}$$

without moving the input pointer, as required.

($\Leftarrow$): We suppose that there is an execution path through the RM-parser which results in *acc*. We shall show that, if there is an execution path through the RM-parser on input $a_1 \ldots a_{n+1}$ which results in the stack

$$0, x_1, h_1, \ldots, x_q, h_q$$

and input pointer pointing at a_{i+1}, then there is a node $v \in U_i$ and a path



in G. Then, since the RM-parser results in *acc*, there is an execution path through the RM-parser which results in h_n, the accept state, on the top of the stack and the input pointer pointing at $a_{n+1} = \$$. So there is a node $v_n \in U_n$ whose label is h_n, and the GRMLR algorithm will report success.

We prove the result by induction on the number of execution steps in the execution path.

When the execution begins, the input pointer points at a_1 and the stack just contains the start state, 0. The GSS has base state $v_0 \in U_0$, and clearly there is a path of length 0 from v_0 to itself.

Now suppose that it takes M execution steps to result in the stack

$$0, x_1, h_1, \ldots, x_{q-1}, h_{q-1}, x_q, h_q$$

and input pointer pointing to a_{i+1}, and that the result is true for execution paths which contain fewer than M steps.

If the last execution step, the one which resulted in the given stack and position, was a shift action, then it must have been sh_q, this action must lie in position (h_{q_1}, a_i) of the RM-table, we must have $x_q = a_i$, and the stack must have been

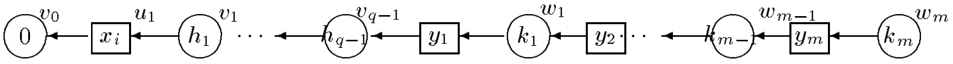
$$0, x_1, h_1, \ldots, x_{q-1}, h_{q-1}$$

By induction, there is a path in the GSS from a node $v_{q-1} \in U_i$ to the node v_0. If $q = 1$ then sh_q is added to Q at the start of the algorithm. Otherwise, sh_q is added to Q when the node v_{q-1} was created either by the REDUCER or the SHIFTER. Then, when SHIFTER(i) is executed, the node v_q and a path of length 2 from v_q to v_{q-1} will be created, as required.

Now suppose that the last execution step was a reduction, so that the stack was of the form

$$0, x_1, h_1, \ldots, x_{q-1}, h_{q-1}, y_1, k_1, \ldots, y_m, k_m$$

with the input pointer pointing at a_{i+1}. Thus (rf, m) lies in position (k_m, a_{i+1}) of the RM-table, where f is $x_q ::= y_1 \ldots y_m \delta$, $\delta \overset{*}{\Rightarrow} \epsilon$, and gh_q lies in position (h_{q-1}, x_q) of the RM-table. By induction, there is a path in the GSS

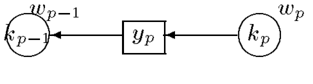


with $w_m \in U_i$.

If $v_{q-1} \in U_i$ then all of the state nodes from w_m to v_{q-1} must also be in U_i. Hence, by Lemma 3, we must have $y_d \overset{*}{\Rightarrow} \epsilon$, for $1 \leq d \leq m$. Since the item

$(x_q ::= y_1 \dots y_m \cdot \delta, a_{i+1})$ lies in k_m, the item $(x_q ::= \cdot y_1 \dots y_m \delta, a_{i+1})$ lies in h_{q-1} and thus $(rf, 0)$ lies in position (h_{q-1}, a_{i+1}) of the RM-table. Looking at the GRMLR algorithm, we see that when the node v_{q-1} was created, the pending action $(v_{q-1}, x_q, 0)$ would have been added to R. Since $v_{q-1} \in U_i$, this action would be removed from R during the i^{th} step of the GSS construction. When this action was removed from R, a node $v_q \in U_i$ labelled h_q and the path v_q, u_q, v_{q-1} would have been added to the GSS if they were not already there. Thus there is a path from v_q to v_0, as required.

Now suppose that $w_{p-1} \notin U_i$ and that $w_d \in U_i$ for $d \geq p$ (here $v_{q-1} = w_0$). We may assume that $1 \leq p \leq m$. By the same reasoning as above we have that (rf, p) lies in position (k_p, a_{i+1}) of the RM-table. If the path



was created by the SHIFTER then, since $w_p \in U_i$, it must have been created by SHIFTER$(i-1)$. Since (rf, p) lies in position (k_p, a_{i+1}) and $p \geq 1$, (w_{p-1}, x_q, p) would have been added to R by SHIFTER$(i-1)$. If this path was created by the REDUCER, it must have been REDUCER(i). For REDUCER(i) to create the above path of length 2 to v_{p-1} it must be processing a reduction of the form (v', x_p, m') and there must be a path of length $2(m'-1)$, or length 0 if $m' = 0$, from v' to w_{p-1}. If $m' = 0$ then we must have $v' \in U_i$ and $w_{p-1} = v'$, which is contrary to the assumption that $w_{p-1} \notin U_i$. Thus we must have $m' \neq 0$ and (w_{p-1}, x_q, p) would have been to added R by REDUCER(i). Thus, in either case, when the element (w_{p-1}, x_q, p) was removed from R for processing, a node $v_q \in U_i$ labelled h_q and the path v_q, u_q, v_{q-1} would have been added to the GSS if they were not already there.

This completes the proof.

6.5 The reduction modified DFA and LR(1) grammars

In this section we shall study the conflicts that can arise in a reduction modified table generated from an LR(1) grammar. We shall show that, as long as the grammar is LR(1), such conflicts can be resolved by removing multiple reductions. As a side effect, if the correct reductions are removed, the resulting algorithm is slightly more efficient in the sense that there is a slight reduction in stack activity.

We begin with two lemmas on the nature of DFA states which contain right nullable items. These lemmas will be required in the subsequent proofs.

Lemma 4 *If a DFA state h contains an item of the form $(A ::= \alpha \cdot D\beta, a)$ where $D\beta \overset{+}{\Rightarrow} \epsilon$, for some non-terminal, Z, $D \underset{rm}{\overset{*}{\Rightarrow}} Z \overset{*}{\Rightarrow} \epsilon$, and for all such Z, h contains the item $(Z ::= \cdot, a)$.*

Proof If $D\beta \overset{+}{\Rightarrow} \epsilon$ then $D \underset{rm}{\overset{p}{\Rightarrow}} \epsilon$, for some $p \geq 1$. If $p = 1$ then we can take $Z = D$. If $p \geq 2$ then we have $D \Rightarrow C\gamma \underset{rm}{\overset{p-1}{\Rightarrow}} \epsilon$, so $C \underset{rm}{\overset{q}{\Rightarrow}} \epsilon$ for some $q \leq p-1$ and, by the construction of DFA states, $(D \cdot C\gamma, a)$ lies in h. Thus, by induction, there exists some Z such that $C \underset{rm}{\overset{*}{\Rightarrow}} Z \Rightarrow \epsilon$. Since $\gamma \overset{*}{\Rightarrow} \epsilon$, we have $D \Rightarrow C\gamma \underset{rm}{\overset{*}{\Rightarrow}} C \underset{rm}{\overset{*}{\Rightarrow}} Z \Rightarrow \epsilon$, as required.

Now suppose that Z is a non-terminal such that $D \underset{rm}{\overset{q}{\Rightarrow}} Z {\Rightarrow} \epsilon$, where $q \geq 0$. If $q = 0$ then $Z = D$ and, by construction of the DFA states and since $\beta \overset{*}{\Rightarrow} \epsilon$, $(D ::= \cdot, a)$ lies in h. Now suppose that $q \geq 1$ and that $D {\Rightarrow} C\gamma \underset{rm}{\overset{q-1}{\Rightarrow}} Z {\Rightarrow} \epsilon$. Since this is a rightmost derivation and Z is a non-terminal, we must have $C \underset{rm}{\overset{k}{\Rightarrow}} Z$, where $k \leq q - 1$. By the DFA construction process, we have that $(D \cdot C\gamma, a)$ lies in h. So, by induction, $(Z ::= \cdot, a)$ lies in h, as required.

6.5.1 The correctness of reduced RM-tables for LR(1) grammars

In this section we shall examine the properties that RM-tables have when the grammar is known to be LR(1), we shall define the concept of a reduced RM-table, and we shall we prove that the table based parser which uses a reduced RM-table defined above is correct for LR(1)-grammars.

We suppose that we have an RM-table which is constructed from an LR(1)-grammar, Γ, so that there are no conflicts in the LR(1)-table.

Lemma 5 *If Γ is an LR(1) grammar then the RM-table for Γ cannot contain any shift/reduce conflicts.*

Proof Suppose that position (h, a) of the RM-table contains a shift/reduce conflict, sk and (rf, m) say. From the RM-table construction we must have that the DFA state h contains items of the form $(X ::= \alpha \cdot a\beta, b)$ and $(Y ::= \tau \cdot \sigma, a)$, where $|\tau| = m$ and $\sigma \overset{*}{\Rightarrow} \epsilon$.

If $\sigma = \epsilon$ then this conflict would also appear in the LR(1)-table, contrary to the assumption that Γ is LR(1).

If $\sigma \neq \epsilon$ then we have $\sigma = B_1\sigma_1$, where $B_1 \overset{+}{\Rightarrow} \epsilon$ and $\sigma_1 \overset{*}{\Rightarrow} \epsilon$. Then, by Lemma 4, h contains an item of the form $(Z ::= \cdot, a)$ and there would be a conflict in the LR(1)-table. Thus the RM-table cannot contain a shift/reduce conflict if Γ is LR(1).

Definition A *reduced RM-table* is a table obtained by removing some, but not all, of the reductions from the entries in the RM-table.

From Lemma 4 we see that if an entry in an RM-table contains a reduction then it contains an LR(1) reduction, thus the LR(1)-table is a reduced RM-table obtained by simple removing all the non-LR(1) reductions. In Section 6.5.2 we shall prove a lemma which describes the relationship between reductions in the same entry of an RM-table and use this to select which reduction to retain, removing all the others so that the table becomes conflict-free. Before we do this we prove the following theorem, which shows that reduced RM-parsers are correct for LR(1)-grammars.

Theorem 3 *If Γ is an LR(1) grammar then the language accepted by a reduced RM-parser for Γ is precisely the language generated by Γ.*

Proof Suppose that the input string u is accepted by a given reduced RM-parser. The reduced RM-table is a subset of the RM-table so the execution path which results in acceptance is also an execution path through the RM-parser.

Then, by Theorem 1, u is accepted by the LR(1)-parser and hence is in the language generated by Γ.

Conversely, suppose that $u = a_1 \dots a_n$ is in the language generated by Γ. Then u is accepted by the LR(1)-parser for Γ. We shall show, by induction on the length of the execution path, that if there is an execution path, Θ say, through the LR(1)-parser on input u which results in the stack

$$0, x_1, h_1, \dots, x_m, h_m$$

and the input pointer pointing at a_{i+1}, then there is a (possibly trivial) extension of this execution path, Θ', which results in the stack

$$0, z_1, l_1, \dots, z_p, l_p$$

and the input pointer pointing at a_{i+1}, such that there is an execution path, Ψ, through a given reduced RM-parser which results in the same stack and input pointer position.

The result follows from this because, since u is accepted by the LR(1)-parser, there is an execution path through this parser which results in a stack with the accept state on top and the input pointer pointing at \$. Since the grammar is LR(1), there is no non-trivial extension of this execution path, so there must be a path through the reduced RM-parser which results in the same stack and input pointer position, hence it will also accept u.

If the execution path through the LR(1)-parser has length 0 then the stack just contains the start state and the input pointer points to a_1. Clearly, the zero length execution path through the reduced RM-parser results in the same stack and input position, thus we can take the trivial extension of the execution path through the LR(1)-parser.

Now suppose that we have a given execution path, Θ, through the LR(1)-parser which results in the stack

$$0, x_1, h_1, \dots, x_m, h_m$$

and the input pointer pointing at a_{i+1}, and that for all execution paths of shorter length there is an extension and a corresponding path through the reduced RM-parser which result a common stack and the input pointer still pointing at a_{i+1}.

If the last step in the given execution path Θ was a shift action then we must have that the execution path, Θ_1, up to the point of this last step resulted in the stack

$$0, x_1, h_1, \dots, x_{m-1}, h_{m-1}$$

with the input pointer pointing at a_i, and that position (h_{m-1}, a_i) of the LR(1)-table contained the action sh_m. By induction there is an extension, Θ'_1, of Θ_1 and an execution path, Ψ_1, through the reduced RM-parser which result in a common stack and the input pointer pointing at a_i. Since the grammar is assumed to be LR(1), the only execution step which can be taken from state h_{m-1} with input a_i is sh_m, which will advance the input pointer. Thus the extension of this path must be the trivial extension, so $\Theta'_1 = \Theta_1$, and so Ψ_1 must result in the stack

$$0, x_1, h_1, \dots, x_{m-1}, h_{m-1}$$

with the input pointer pointing at a_i. The shift actions in the reduced RM-table correspond exactly to the shift actions in the LR(1)-table, so position (h_{m-1}, a_i) of the reduced RM-table contains the action sh_m. Thus Ψ_1 can be extended to result in the stack

$$0, x_1, h_1, \ldots, x_{m-1}, h_{m-1}, a_i = x_m, h_m$$

and the input pointer pointing at a_{i+1}.

Finally suppose that the last step in the given execution path through the LR(1)-parser was a reduce action, so we must have that the execution path, Θ_1, up to the point of this last step resulted in the stack

$$0, x_1, h_1, \ldots, x_{m-1}, h_{m-1}, y_1, k_1, \ldots, y_q, k_q$$

with the input pointer pointing at a_{i+1}, that position (k_q, a_{i+1}) contained the action rj, where j is the rule $x_m ::= y_1 \ldots y_q$, and that position (h_{m-1}, x_m) of the LR(1)-table contained the action gh_m. Position (k_q, a_{i+1}) of the RM-table for Γ contains the action (rj, q), so this entry in the reduced RM-table must contain an action of the form (ri, t) where i is a rule of the form $Z ::= \gamma\delta$, $|\gamma| = t$, and $\delta \overset{*}{\Rightarrow} \epsilon$.

Now, by induction there is an extension, Θ_1', of Θ_1 and an execution path, Ψ_1, through the reduced RM-parser which result in a common stack and the input pointer pointing at a_{i+1}. If this extension is non-trivial (i.e. includes at least one additional execution step) then, since the grammar is LR(1), Θ_1' is also an extension of Θ, and we can take $\Psi = \Psi_1$, giving the result. Thus we assume that $\Theta_1' = \Theta_1$ and thus that Ψ_1 results in the stack

$$\begin{aligned} & 0, x_1, h_1, \ldots, x_{m-1}, h_{m-1}, y_1, k_1, \ldots, y_q, k_q \\ & \qquad = \quad 0, z_1, l_1, \ldots, z_{p-1}, l_{p-1}, w_1, g_1, \ldots, w_t, g_t \end{aligned}$$

with the input pointer pointing at a_{i+1}. We can extend Ψ_1 to Ψ by performing the reduction (ri, t) which results in the stack

$$0, z_1, l_1, \ldots, z_{p-1}, l_{p-1}, Z, l_p$$

where position (l_{p-1}, Z) of the reduced RM- (and the LR(1)-) table contains the action gl_p. By Lemma 1 there is an extension, Θ', of Θ_1 which results in the same stack. Since Γ is LR(1) there is a unique next execution step from each execution path through the LR(1)-parser, thus Θ' must also be an extension of Θ, as required.

Note The above theorem actually shows that although the RM-table for an LR(1) grammar may contain conflicts it doesn't matter which of the conflicting reductions is chosen, the parse will be successful if the input string is in the language.

6.5.2 Resolved RM-parse tables

In this section we shall describe the relationship between reductions in the same entry of an RM-table, and show that there is a base reduction which is generated by the other reductions. We shall then give an example which compares an LR(1)-parser with the (deterministic) reduced RM-parser obtained by retaining the base reductions and removing all other conflicting reductions.

Lemma 6 *Suppose that Γ is an LR(1) grammar and that h is a state in the LR(1) DFA for Γ which contains an RM-table reduce/reduce conflict, then h contains an item of the form $(Y ::= \tau \cdot C\sigma, a)$, called a* base item, *such that $C\sigma \overset{+}{\Rightarrow} \epsilon$ and for all other items $(Z ::= \gamma \cdot \delta, a)$ in h, with $\delta \overset{*}{\Rightarrow} \epsilon$, $C \overset{*}{\underset{rm}{\Rightarrow}} Z$ and $\gamma = \epsilon$.*

Proof Suppose that the DFA state h contains two items of the form $(X ::= \alpha \cdot \beta, a)$ and $(Z ::= \gamma \cdot \delta, a)$, where $\beta \overset{m}{\underset{rm}{\Rightarrow}} \epsilon$ and $\delta \overset{n}{\underset{rm}{\Rightarrow}} \epsilon$. If $m = n = 0$ then Γ is not LR(1), thus without loss of generality we may suppose that $m \geq 1$. We shall prove, by induction on $m + n$, that $\gamma = \epsilon$ and that $\beta = B\tau$, where $B \overset{*}{\underset{rm}{\Rightarrow}} Z$. We shall then show that the result follows from this.

Since $m \geq 1$ we have $\beta = B\tau$ where $B \overset{+}{\Rightarrow} \epsilon$. Also, by Lemma 4, there is a non-terminal D such that $B \overset{*}{\underset{rm}{\Rightarrow}} D$ and $(D ::= \cdot, a)$ lies in h. Furthermore, either $B = D$ or there is some non-terminal E such that $B \overset{*}{\underset{rm}{\Rightarrow}} E \Rightarrow D\nu$, where $\nu \overset{q}{\underset{rm}{\Rightarrow}} \epsilon$ and $0 \leq q \leq m$. Then the item $(E ::= \cdot D\nu, a)$ lies in h.

If $n = 0$ then $\delta = \epsilon$ and so, since the grammar is LR(1), to avoid a reduce/reduce conflict in h we must have $Z = D$ and $\gamma = \epsilon$. Thus the result is true for $m + n = 1$, and for all values of $m + n$ when $n = 0$.

Now suppose that $n \geq 1$, and that for any DFA state k which contains two items $(X' ::= \alpha' \cdot \beta', a')$ and $(Z' ::= \gamma' \cdot \delta', a')$, where $\beta \overset{m'}{\underset{rm}{\Rightarrow}} \epsilon$, $\delta \overset{n'}{\underset{rm}{\Rightarrow}} \epsilon$, $m' \geq 1$ and $m' + n' < m + n$, then $\gamma' = \epsilon$ and $\beta' = B'\tau'$, where $B' \overset{*}{\underset{rm}{\Rightarrow}} Z'$.

Since $n \geq 1$ we have $\delta = C\sigma$ where $C \Rightarrow \mu \overset{p}{\underset{rm}{\Rightarrow}} \epsilon$, and $p < n$. By construction of the DFA states we have that $(C ::= \cdot\mu, a)$ lies in h and $m + p < m + n$, so, by induction, $B \overset{*}{\underset{rm}{\Rightarrow}} C$ and $\mu = \epsilon$. But then, since Γ is LR(1) and $(C ::= \cdot, a)$ and $(D ::= \cdot, a)$ both lie in h, we must have $C = D$.

From the DFA construction, there is a state k which can be reached from h via a transition labelled D, and k contains $(Z ::= \gamma D \cdot \sigma, a)$ and $(X ::= \alpha D \cdot \tau, a)$, if $B = D$, or $(E ::= D \cdot \nu, a)$. Furthermore, $\sigma \overset{n'}{\underset{rm}{\Rightarrow}} \epsilon$, $\tau \overset{m'}{\underset{rm}{\Rightarrow}} \epsilon$ and $\nu \overset{q}{\underset{rm}{\Rightarrow}} \epsilon$, where $n' < n$ and $m', q \leq m$.

If $(E ::= D \cdot \nu, a) \in h)$ and $n' + q \geq 1$, then by induction either D or γD must equal ϵ. This is a contradiction. Thus we must have $n' = q = 0$ and $\nu = \sigma = \epsilon$. But then, since Γ is LR(1), we must have $E = Z$ and $\gamma = \epsilon$, so $B \overset{*}{\underset{rm}{\Rightarrow}} Z$, as required.

If $(X ::= \alpha D \cdot \sigma, a) \in k$ then, again, by induction we must have $n' = m' = 0$ and $\tau = \sigma = \epsilon$. In this case, since we assumed that the original items were distinct, k contains an LR(1) reduce/reduce conflict, contrary to the assumption that Γ is LR(1).

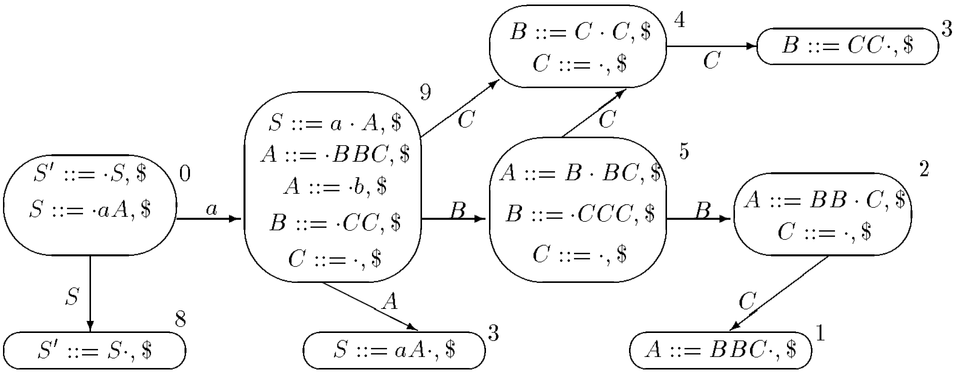
Finally we need to show that h contains an RM-table reduce/reduce conflict then h contains a base item, as defined in the statement of the lemma. If h contains two distinct items of the forms $(Y ::= \tau \cdot \mu, a)$ and $(X ::= \alpha \cdot \beta, a)$, where $\mu \overset{*}{\Rightarrow} \epsilon$ and $\beta \overset{*}{\Rightarrow} \epsilon$, then, since Γ is LR(1) we may assume that $\mu \neq \epsilon$. So $\mu = C\sigma \overset{+}{\Rightarrow} \epsilon$.

But then by the above argument, for any other item $(Z ::= \gamma \cdot \delta, a)$ in h, we must have $\gamma = \epsilon$ and $C \overset{*}{\underset{rm}{\Rightarrow}} Z$, as required.

Definition If Γ is an LR(1) grammar we define *resolved RM-table* for Γ to be the reduced RM-table obtained by taking the RM-table for Γ and resolving the reduce/reduce conflicts, if there are any, by selecting a base item $(X ::= \alpha \cdot \beta, a)$ and removing all other reductions in that entry of the table.

By Theorem 3 we see that a resolved RM-parser is correct for any LR(1) grammar. A resolved RM-parser is also clearly deterministic for LR(1) grammars. Finally, although we do not show it here it is not hard to see that the resolved RM-parser is the most efficient reduced RM-parser in the sense that it induces the least stack activity. We illustrate this with the following example.

$$\begin{array}{lcl} S' & ::= & S \\ S & ::= & aA \mid \epsilon \\ A & ::= & BBC \mid b \\ B & ::= & CC \\ C & ::= & \epsilon \end{array}$$



RM-parse table

	\$	a	b	A	B	C	S
0	(r2,0)	s9					g9
1	(r3,3)						
2	(r6,0) (r3,2)					g1	
3	(r5,2)						
4	(r6,0) (r5,1)					g3	
5	(r6,0) (r5,0) (r3,1)				g2	g4	
6	(r1,2)						
7	(r4,1)						
8	acc						
9	(r6,0) (r5,0) (r3,0) (r1,1)		s7	g6	g5	g4	

LR(1)-parse table

	$	a	b	A	B	C	S
0	r2	s9					g9
1	r3						
2	r6					g1	
3	r5						
4	r6					g3	
5	r6				g2	g4	
6	r1						
7	r4						
8	acc						
9	r6		s7	g6	g5	g4	

resolved RM-parse table

	$	a	b	A	B	C	S
0	(r2,0)	s9					g9
1	(r3,3)						
2	(r3,2)					g1	
3	(r5,2)						
4	(r5,1)					g3	
5	((r3,1)				g2	g4	
6	(r1,2)						
7	(r4,1)						
8	acc						
9	(r1,1)		s7	g6	g5	g4	

We use the LR(1)-table and the resolved RM-table to parser the input string a, showing that the resolved RM-parser requires much less stack activity.

LR(1)-table resolved RM-table

stack	input pointer	next action		stack	input pointer	next action
0	a	s9		0	a	s9
0a9	$	r6		0a9	$	(r1,1)
0a9C4	$	r6		0S8	$	acc
0a9C4C3	$	r5				
0a9B5	$	r6				
0a9B5C4	$	r6				
0a9B5C4C3	$	r5				
0a9B5B2	$	r6				
0a9B5B2C1	$	r3				
0a9A6	$	r1				
0S8	$	acc				

7 Experimental results

We have implemented Algorithm 1e and our GRMLR algorithm (with the slight modification described in Section 6.3), and we have run these algorithms on several test grammars. In this section we shall describe these experiments and discuss the results.

The aim of this report is to explore the theoretical aspects of Tomita-style GLR algorithms in a way which allows feed-back into improvements in the algorithms and which illuminates implementation. Our discussion in this section is intended only to illustrate the effects of our modifications with explicit examples. Thus we shall not discuss here the actual implementations or the many implementation issues which needed to be addressed. However, the implementations form part of the GTB toolset which can be downloaded from

```
http://www.cs.rhul.ac.uk/research/languages/index.shtml
```

The GTB input and output files for the examples discussed in this section can be found in the GTB version 1.0 distribution.

7.1 The experiments

We have run both Algorithm 1e and our GRMLR algorithm on a grammar for ANSI-C, a grammar for (a slightly extended) level 0 ISO-Pascal, and on four 'toy' grammars which feature right-nullable rules. In all cases we have used the LR(1) table as the basis of the algorithms, but our parser generator tool (GTB) can be configured to generate and use LR(0), SLR(1) or LALR tables if preferred.

The aims of our experiments were firstly to test the practicality of the Tomita approach, secondly to test the effects of our GRMLR algorithm, and thirdly to compare our GRMLR algorithm with Tomita's original algorithm.

The first aim was addressed by using grammars for the real languages ANSI-C and Pascal. In both cases the grammars were specified in BNF, because at the moment our generalised parser generators have not been extended to accept EBNF. We were interested in the sizes of the graph structured stacks generated and the amount of effort required to build them. Thus the output of the experiments includes the number of nodes of each type in the final GSS, and the number of times each of these nodes is visited during the construction. For both C and Pascal the parsers generated were run on a large input program file.

The second and third aims were combined and then de-composed into two parts: checking that the GRMLR algorithm could handle right-nullable grammars, and considering the potential decrease in efficiency that might be introduced because of the increased number of conflicts in the underlying parse table.

To look at the effect of the two algorithms on right-nullable grammars we looked at four small examples. The first two

$$\begin{array}{lcl} S & ::= & aAAA \mid \epsilon \\ A & ::= & a \mid \epsilon \end{array} \qquad\qquad \begin{array}{lcl} S & ::= & aSAAA \mid \epsilon \\ A & ::= & a \mid \epsilon \end{array}$$

contain a relatively large amount of right-nullability and ambiguity. The second also contains hidden right recursion. However, both of them are correctly parsable using Algorithm 1e. The third,

$$\begin{array}{lcl} S & ::= & aDad \mid BDab \\ D & ::= & aAB \\ A & ::= & aBB \mid \epsilon \\ B & ::= & \epsilon \end{array}$$

is parsable with Algorithm 1e if the nodes in the U_i are processed in a certain order but not if they are processed in a different order, see Section 4.5. Our implementation of Algorithm 1e has been set up to process the nodes in an order which shows the failure. (Note, this grammar does not contain hidden right recursion.) The fourth grammar, below, is not parsable by Algorithm 1e at all, see Section 4.4.

$$\begin{array}{lcl} S & ::= & bA \\ A & ::= & aAB \mid \epsilon \\ B & ::= & \epsilon \end{array}$$

As for the potential increase in efficiency of our GRMLR algorithm, the concern comes from the fact that there are more conflicts in the parse table (although no more states) and that this may lead to more states in a given U_i in the GSS, and hence to a larger and less efficient structure. In fact we claim that the GSS constructed by our (slightly modified) method is identical to the GSS constructed by Algorithm 1e in the cases where Algorithm 1e works. Although there are more conflicts in the table, for a given input string the same number of conflicts are encountered, it is just that some of them are encountered earlier in the stack construction process. Furthermore, when right-nullable reductions are applied only the non-nullable left portion of the rule is retraced, so the length of path traversed in the GSS when performing such a reduction is less than for Algorithm 1e. So our construction method is in fact slightly more efficient! We illustrate these effects by counting the numbers of nodes and edges in the GSSs constructed by each algorithm, and observing that they are identical, and by counting the number of node visits made by each algorithm and observing that in cases where there are right nullable rules this number is lower for our GRMLR algorithm than for Algorithm 1e.

All our experiments have been run on a 400MHz Pentium II processor with 128Mbyte RAM using GTB version 2.00 compiled using Borland C version 5.1

7.2 The results

Experiment 1: ANSI-C

Aim: Parse the source code for Quine-McCluskey minimiser ‘bool’ [Joh93] using ANSI-C grammar from [KR88].

The grammar is modified to allow the lexer primitive $\boxed{\text{string}}$ to match one or more ‘STRING’ keywords so as to model ANSI-C automatic string concatenation which is usually handled in the lexer.

Preparation:

1. Remove preprocessor lines and comments from bool source by passing through the Borland 5.1 standalone preprocessor.

```
cpp32 -P- bool.c
```

2. The preprocessor leaves #pragma lines in the expanded code marking file inclusion boundaries. A side-effect of the preprocessor is that many blank lines are left in expanded source code.

Manually remove #pragma lines (8) and blank lines from bool.i

3. Borland C version 5.1 standard library headers contain some non-ANSI-C syntax.

Manually remove lines from Borland header files: leaving 796 lines

4. Produce 'lexicalised' source by replacing integer constants with the keyword INTEGER, identifiers by ID etc, using an RDP generated C pretty printer with -L (lexicalise) option.

```
pretty_c -L bool.i > bool.str
```

796 lines, 4921 tokens, average of 5.39 tokens per line. (Note: original source shows 3.83 tokens per line showing effects of comment and blank line removal.)

5. Create C_tomita.gtb and C_null.gtb each with ten calls to the tomita_1 (Algorithm 1e) and tomita_1_nullable_accepts (GRMLR) parsers respectively.

Experimental runs:

The first run below is the Algorithm 1e running on the ANSI C grammar, the second run is the GRMLR algorithm.

```
gtb -T70000 C_tomita.gtb
lr(1) parse table requires 285723 cells: 421 cells have conflicts

gtb -T70000 C_null.gtb
lr(1) parse table requires 285723 cells: 421 cells have conflicts
```

Run times, showing non-linearity in memory allocation subsystem:

run no.	1	2	3	4	5	6	7	8	9	10
time	1.18	1.19	1.18	1.18	1.19	1.36	1.52	1.86	2.02	3.03

Each run of the parser creates a new copy of the GSS data structure each of which contains more than 100,000 nodes and edges. After the fifth run we see that parse times begin to increase. This effect is generated by non-linearities in the memory allocator for Borland-C: heap fragmentation has been observed to generate at least quadratically increasing allocation times in other applications once a certain level of allocation has been reached. We should stress that this behaviour is not a side effect of page swapping since system monitoring shows that the whole set of GSS's fits into physical memory and no swapping occurs.

Tomita-style parsers must, of course, necessarily create large structures when parsing long strings. The rather few reports of Tomita-parsing performance in the literature usually ignore the impact of allocating such structures on the parser run time, possibly because typical Tomita applications are in natural language parsing where the strings are often very short by our standards.

Although this table illustrates difficulties with the Borland-C allocater, it is not unusual for other C runtime libraries to display such behaviour. In a better behaved implementation, we would preallocate memory based on string length. In the rest of this report, we have selected experiments in which the GSS's never grow large enough to trigger this behaviour.

The first table below shows the size of the GSS generated by each of the algorithms. We note that the numbers are the same for each algorithm. The second and third tables show the amount of effort involved in constructing the GSS in terms of the number of times each node and edge is visited. We list the number of nodes visited 0 times (those which are constructed but not visited again) and those visited 1 and more times. For the C grammar these numbers are the same for both algorithms because the ANSI-C grammar does not contain any nullable non-terminals, but for our other examples we shall see that the number of visits is lower for the GRMLR algorithm.

Size of the GSS

	levels	state nodes	shift nodes	reduce nodes	edges
Algthm1e	4292	28323	4496	23962	56935
GRMLR	4292	28323	4496	23962	56935

Node visit counts

visits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	31	32	33	34	35	36	42	44
no. nodes Algthm1e	24112	30026	606	266	126	128	171	90	32	98	56	56	40	34	3	13	36	41	89	234	170	61	199	40	24	1	1	9	8	3	1	1	1	1	1	1	2	1
no. nodes GRMLR	24112	30026	606	266	126	128	171	90	32	98	56	56	40	34	3	13	36	41	89	234	170	61	199	40	24	1	1	9	8	3	1	1	1	1	1	1	2	1

Edge visit counts

visits	0	1	2
no. edges Algthm1e	383	55987	566
no. edges GRMLR	383	55987	566

Experiment 2: Pascal

Aim: Parse the source code for the str21 string preprocessor [Joh91] using the ISO-standard Pascal grammar. The grammar is modified to allow some Borland Turbo-Pascal extensions, and converted from EBNF to BNF using the ebnf2bnf tool which is part of the gtb toolset.

Preparation:

1. Convert pascal.bnf from the RDP v1.5 distribution to BNF using the `ebnf2bnf` tool.

```
ebnf2bnf pascal.bnf -opascal.gtb
```

2. Remove empty production comment ::= . from pascal.gtb.

3. Produce 'lexicalised' source by running the RDP Pascal syntax checker with -L option. Note that this step removes comments.

```
pascal -L str21.pas >str21.str
```

leaving 267 lines, 1829 tokens average of 6.85 tokens per line.

4. Create P_tomita.gtb and P_null.gtb each with ten calls to the tomita_1 and tomita_1_nullable_accepts parsers respectively.

Experimental runs:

```
gtb -T70000 P_tomita.gtb
lr(1) parse table requires 258075 cells: 6 cells have conflicts

gtb -T70000 P_null.gtb
lr(1) parse table requires 258075 cells: 1123 cells have conflicts
```

The run times are 0.39s in each case since the GSS is not large enough to trigger non-linearity in the memory allocation system.

Size of the GSS

	levels	state nodes	shift nodes	reduce nodes	edges
Algthm1e	1830	9171	1861	7373	18469
GRMLR	1830	9171	1861	7373	18469

Node visit counts

visits	0	1	2	3	4	5	6	7	8	9
no. nodes Algthm1e	4834	12134	455	278	188	64	68	195	149	30
no. nodes GRMLR	9538	7468	418	277	188	64	69	194	149	30

visits	10	13	14	15	20	24	25	26	27	total
no. nodes Algthm1e	1	2	2	1	2	0	1	0	2	18383
no. nodes GRMLR	1	2	2	1	2	1	0	1	1	13637

Edge visit counts

visits	0	1	2	4	13	14
no. edges Algthm1e	219	18169	76	2	2	2
no. edges GRMLR	4959	13435	70	2	2	2

Note the increase in conflicts as a result of adding right-nullable accept states. It turns out that the Pascal grammar has many more right-nullable rules than the C grammar. These rules exist because it is legal to have declarations which can optionally be followed by an assignment, const declarations which are only optionally followed by an actual declaration, etc. These do not generate right-nullable rules in the C grammar because in C the semi-colon is a statement terminator and thus is included at the end of such rules, while in Pascal semi-colon is a statement separator, so it is included higher up in the grammar.

The GRMLR algorithm displays 13637 node visits compared to the 18383 node visits for Algorithm 1e.

Experiment 3: A right nullable grammar

Aim: To compare the Algorithm 1e and GRMLR algorithm generated graph structured stacks on a grammar with a large proportion of ambiguity and right nullability. Tests run on strings of lengths 1 to 4.

$$\begin{array}{lll} S & ::= & aAAA \mid \epsilon \\ A & ::= & a \mid \epsilon \end{array} \qquad \textit{input strings}: \; a, \; aa, \; aaa, \; aaaa$$

```
gtb -T70000 P_tomita.gtb
lr(1) parse table requires 56 cells: 2 cells have conflicts

gtb -T70000 P_null.gtb
lr(1) parse table requires 56 cells: 6 cells have conflicts
```

Size of the GSS

	string	levels	state nodes	shift nodes	reduce nodes	edges
Algthm1e	a	2	6	1	4	10
GRMLR	a	2	6	1	4	10
Algthm1e	aa	3	10	3	8	23
GRMLR	aa	3	10	3	8	23
Algthm1e	aaa	4	13	5	9	29
GRMLR	aaa	4	13	5	9	29
Algthm1e	$aaaa$	5	14	6	8	29
GRMLR	$aaaa$	5	14	6	8	29

Node visit counts

	a		*aa*					*aaa*					*aaaa*		
visits	0	1	0	1	2	3	4	0	1	2	3	4	0	1	2
no. nodes Algthm1e	3	8	5	10	2	3	1	10	10	3	3	1	16	9	3
no. nodes GRMLR	9	2	9	6	2	3	1	12	8	3	3	1	16	9	3

Edge visit counts

	a		*aa*				*aaa*				*aaaa*	
visits	0	1	0	1	2	3	0	1	2	3	0	1
no. edges Algthm1e	2	8	2	17	2	2	6	19	2	2	14	15
no. edges GRMLR	8	2	6	13	2	2	8	17	2	2	14	15

Note, there is no ambiguity for the string of length 4, the number of node visits has significantly reduced and the two algorithms perform the same number of visits. For the other three strings the total number of (non-zero) node visits is less for the GRMLR algorithm than for Algorithm 1e.

Experiment 4: A right recursive grammar

Aim: To compare the Algorithm 1e generated and GRMLR algorithm generated graph structured stacks on a grammar with hidden right recursion and a high proportion of ambiguity. Tests run on strings of lengths 1 to 8.

$$\begin{array}{lcl} S & ::= & aSAAA \mid \epsilon \\ A & ::= & a \mid \epsilon \end{array} \qquad \textit{input strings}: \ a, \ aa, \ ..., \ aaaaaaaa$$

```
gtb -T70000 P_tomita.gtb
lr(1) parse table requires 98 cells: 7 cells have conflicts

gtb -T70000 P_null.gtb
lr(1) parse table requires 98 cells: 16 cells have conflicts
```

For this example, for each input string we shall just give the sizes of the GSS's, to show that they are identical for both algorithms, and the total numbers of edge visits, to show that these are lower for the GRMLR algorithm than for Algorithm 1e.

Size of the GSS

	string	levels	state nodes	shift nodes	reduce nodes	edges
Algthm1e	a	2	7	1	5	12
GRMLR	a	2	7	1	5	12
Algthm1e	a^2	3	17	4	15	39
GRMLR	a^2	3	17	4	15	39
Algthm1e	a^3	4	27	7	28	75
GRMLR	a^3	4	27	7	28	75
Algthm1e	a^4	5	37	10	42	113
GRMLR	a^4	5	37	10	42	113
Algthm1e	a^5	6	47	13	57	153
GRMLR	a^5	6	47	13	57	153
Algthm1e	a^6	7	57	16	73	195
GRMLR	a^6	7	57	16	73	195
Algthm1e	a^7	8	67	19	90	239
GRMLR	a^7	8	67	19	90	239
Algthm1e	a^8	9	77	22	108	285
GRMLR	a^8	9	77	22	108	285

Node visit counts

	a	a^2	a^3	a^4	a^5	a^6	a^7	a^8
total node visits Algthm1e	10	43	103	181	287	425	595	797
total node visits GRMLR	2	35	91	169	275	413	583	785

Experiment 5: A grammar on which Algorithm 1e may break

Aim: To construct a non-hidden-right-recursive grammar on which Algorithm 1e incorrectly rejects some strings if the frontier nodes are processed in a certain order, and to show that the GRMLR algorithm correctly accepts such strings.

$$\begin{array}{rcl} S & ::= & aDad \mid BDab \\ D & ::= & aAB \\ A & ::= & aBB \mid \epsilon \\ B & ::= & \epsilon \end{array}$$

$input\ string : aaab$

```
gtb -T70000 P_tomita.gtb
lr(1) parse table requires 176 cells: 2 cells have conflicts

Tomita 1 parse : 'a a a b'
Reject

gtb -T70000 P_null.gtb
lr(1) parse table requires 176 cells: 5 cells have conflicts
```

```
Tomita 1 parse with nullable accepts: 'a a a b'
Accept
```

In this example, Algorithm 1e rejects the input string $aaab$, whereas it is accepted by the GRMLR algorithm. For completeness we have given the statistics on the sizes of the GSS and the number of node visits, but of course the numbers for the Algorithm 1e generated GSS are not particularly interesting because it does not complete the parse.

Size of the GSS

	levels	state nodes	shift nodes	reduce nodes	edges
Algthm1e	4	17	7	10	34
GRMLR	5	21	9	12	42

Node visit counts

visits	0	1	2	3	total
no. nodes Algthm1e	17	16	1	0	18
no. nodes GRMLR	28	11	2	1	18

Note that the node numbers in the Algorithm 1e GSS are lower than for the GRMLR algorithm in this case because Algorithm 1e does complete the GSS construction.

Experiment 6: A grammar on which Algorithm 1e always breaks

Aim: To construct a grammar on which Algorithm 1e incorrectly rejects some strings no matter what order the frontier nodes are processed in, and to show that the GRMLR algorithm correctly accepts such strings.

$$\begin{array}{rcl} S & ::= & bA \\ A & ::= & aAB \mid \epsilon \\ B & ::= & \epsilon \end{array} \qquad \qquad \textit{input string} : baa$$

```
gtb -T70000 P_tomita.gtb
lr(1) parse table requires 63 cells: 0 cells have conflicts

Tomita 1 parse : 'b a a'
Reject

gtb -T70000 P_null.gtb
lr(1) parse table requires 63 cells: 3 cells have conflicts

Tomita 1 parse with nullable accepts: 'b a a'
Accept
```

Again in this example, Algorithm 1e rejects the input string baa, whereas it is accepted by the GRMLR algorithm.

Size of the GSS

	levels	state nodes	shift nodes	reduce nodes	edges
Algthm1e	4	6	3	3	12
GRMLR	4	8	3	5	16

Node visit counts

visits	0	1	2	total
no. nodes Algthm1e	6	6	0	6
no. nodes GRMLR	8	6	2	10

References

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

[Joh91] Adrian Johnstone. `string21` – a string prepreprocessor for `asm21`. Technical Report CSD–TR–91–8, Royal Holloway, University of London, Computer Science Department, 1991.

[Joh93] Adrian Johnstone. `bool` – a boolean function minimiser. Technical Report CSD–TR–93–25, Royal Holloway, University of London, Computer Science Department, 1993.

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

[Lan74] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Lanugages and Programming: 2nd Colloquium*, Lecture Notes in computer science, pages 255 – 269. Springer-Verlag, 1974.

[NF91] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR parsing*, pages 61–75. Kluwer Academic Publishers, Netherlands, 1991.

[NS96] Mark-Jan Nederhof and Janos J. Sarbo. Increasing the applicability of LR parsing. In H.Bunt and M. Tomita, editors, *Recent advances in parsing technology*, pages 35–57. Kluwer Academic Publishers, Netherlands, 1996.

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

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