Generalised Regular Parsers

Adrian Johnstone and Elizabeth Scott

Royal Holloway, University of London
{A.Johnstone,E.Scott}@rhul.ac.uk

Abstract. Aycock and Horspool have given an algorithm which improves the efficiency of GLR parsers. A grammar is 'reduced' so that there is no recursion apart from non-hidden left recursion, an FA recogniser is then constructed and a stack is used when the recursive parts of the original grammar are required. Aycock and Horspool then give an algorithm which performs all possible traversals of the resulting PDA on a given input string. This mirrors the approach taken by Tomita to perform all traversals of an LR(0) FA. However, Aycock and Horspool's algorithm does not terminate in the case where the grammar contains hidden left recursion. In this paper we give a different method for constructing an FA which recognises the language generated by the grammar provided that the only recursion in the grammar arises from left or right recursion. Using this FA allows us to reduce the number of places that the stack is required. We also give a different algorithm for constructing all traversals of the final PDA which is correct in all cases, including grammars with hidden left recursion. Thus we can apply our algorithm to all context free grammars.

1 Introduction

It is well known that a language is regular if and only if it is defined by a one-way deterministic finite automaton (FA) (see for instance [1], pp 118–120) and that the context free languages are similarly defined by the one-way nondeterministic pushdown automata (PDA). Intuitively, the context free languages include those with properly nested bracket structures. A deterministic FA is unable to guarantee that brackets are paired correctly (the slogan has it that 'regular expressions can't count') but the addition of a stack enables correct nesting to be tested.

The Chomsky hierarchy shows that a regular language may be described by a Context Free Grammar (CFG) and, although it is in general undecidable whether a particular CFG generates a regular language it is useful to think informally of CFGs as having some productions which are regular and some which are necessarily context free. Left and right recursion ($A \stackrel{*}{\Rightarrow} A\beta$, $A \stackrel{*}{\Rightarrow} \alpha A$) produce iterated constructs in the generated language which may be described by regular productions. The context free parts of the language arise from those productions which contain *embedded* recursion such as $A \stackrel{*}{\Rightarrow} \gamma A \delta$ where γ and δ are markers for left and right hand bracketing constructs. In detail, of course, embedded recursion may not be immediately obvious since the recursion may be

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 232–246, 2003.

indirect. To further complicate matters, embedded recursion may be intertwined with left or right recursion.

The usefulness of separating out iterative language constructs from necessarily recursive ones is clearly demonstrated by the widespread use of *extended* context free grammars [2] which directly support the use of regular expressions in rules. In recursive descent parsers it is usual to parse these regular expressions using iteration.

It is reasonable to ask if we can algorithmically discover the context free core of a grammar, automatically replacing left and right recursion with regular expressions, providing an optimal BNF to EBNF conversion. The motivation here is to reduce stack activity during the parse and thus speed up the parser. Although such a technique would be of general application (allowing recursive descent parsers to replace recursive function calls with iteration, for instance) it is particularly interesting to analyse the behaviour of LR shift-reduce parsers from this vantage point since the LR languages correspond to those defined by deterministic PDA's. Regular languages, in the form of the deterministic handle-finding automaton are at the heart of the standard LR parsing scheme. A stack is used to handle self-embedded recursion and right recursion. Interestingly, left recursion is absorbed into the handle finding automaton. In the case of general (non-deterministic) CFG's multiple stacks may be needed to keep track of multiple putative derivations.

Tomita [3] gave an algorithm for recognising any context free language by maintaining a compact representation of these multiple stacks in a 'graph structured stack', allowing all traversals of an LR FA to be performed together. (Note, Tomita's basic algorithm does not work for all context free *grammars* since the grammars must have had their cycles and, in some cases, their ϵ-productions removed. Tomita modified his algorithm to handle ϵ-rules but this algorithm still fails on hidden right recursion. An inelegant fix was subsequently provided by Farshi [4]. We have described elsewhere a modification of Tomita's basic algorithm which works for all context free grammars [5] which is more efficient than the Tomita and Farshi variants.)

Aycock and Horspool [6,7] have given a parsing method in which a grammar is 'reduced' so that there is no recursion apart from non-hidden left recursion. An FA recogniser for the reduced grammar is then constructed, and a stack is used when the recursive parts of the original grammar are required. Aycock and Horspool also give an algorithm which performs all possible traversals of the resulting PDA on a given input string. This mirrors the approach taken by Tomita to perform all traversals of an LR FA.

Aycock and Horspool's algorithm does not terminate in the case where the grammar contains 'hidden' left recursion. This is also the case for Tomita's algorithm but the reasons for the failure to terminate in the two cases are different. Tomita's algorithm really failed on hidden right recursion but he introduced a modification to correct the problem. It was this modification which in turn failed with respect to hidden left recursion (for further discussion of this issue see [8]). In Aycock and Horspool's case if a grammar non-terminal, A say, has

the property that $A \stackrel{*}{\Rightarrow} \alpha A \beta$, where $\alpha \neq \epsilon$ then the automaton which recognises the language generated by A calls itself recursively. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$ (that is, the grammar contains hidden left recursion) then the automaton will repeatedly call itself without consuming any input, and hence will fail to terminate.

In this paper we describe the construction of a *reduction incorporated automaton* from a grammar which we can prove (see [9]) correctly recognises the language generated by the grammar provided that the only recursion in the grammar essentially arises from left or right recursion. Using this automaton rather than the one constructed by Aycock and Horspool allows us to reduce the number of places that the stack is required in the final algorithm. We give a new algorithm for constructing all traversals of the final PDA and we can prove that this algorithm is correct in all cases, including grammars with hidden left recursion. Thus we can apply our modified parsing method to all context free grammars. The theorems in this paper are stated without proof, but full formal proofs can be found in the techincal report [9] on which this paper is based.

2 Initial Definitions

A *context free grammar* consists of a set $\mathbf{N}$ of non-terminal symbols, a set $\mathbf{T}$ of terminal symbols, an element $S \in \mathbf{N}$ called the start symbol, and a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a (possibly empty) string of terminals and non-terminals. We assume that there is an augmented start rule, $S' ::= S$, so that S' does not appear on the right hand side of any grammar rule.

A *derivation step* is an element of the form $\gamma A \delta \Rightarrow \gamma \alpha \delta$ where γ and δ are strings of terminals and non-terminals and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence of derivation steps $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$. We write $\sigma \stackrel{*}{\Rightarrow} \tau$ and $\sigma \stackrel{+}{\Rightarrow} \tau$ if $n > 0$.

A *sentential form* is any string α such that $S \stackrel{*}{\Rightarrow} \alpha$ and a *sentence* is a sentential form which contains only elements of $\mathbf{T}$. The set, $L(\Gamma)$, of sentences which can be derived from the start symbol of a grammar Γ, is defined to be the *language* generated by Γ.

A string α is *nullable* if $\alpha \stackrel{*}{\Rightarrow} \epsilon$ and *null* if $\alpha \stackrel{*}{\Rightarrow} u \in \mathbf{T}^*$ implies that $u = \epsilon$. We say that a grammar has *left (*or *right) recursion* if there is a non-terminal A and a derivation $A \stackrel{*}{\Rightarrow} \alpha A \beta$ where α is nullable (or β is nullable). We say that the recursion is *hidden* if $\alpha \neq \epsilon$ (or $\beta \neq \epsilon$). A grammar has *proper self embedding* if there is some non-terminal, A, and non-null strings α, β such that $A \stackrel{*}{\Rightarrow} \alpha A \beta$.

A *finite automaton* (FA) consists of a set of states and a set of transitions between these states. One of the states is singled out to be the *start* state, and one or more states are designated as *accepting* states. The transitions are labelled with grammar symbols together with the empty string ϵ. For technical reasons we shall want to label some of the transitions with special versions of ϵ which correspond to 'performing a reduction by rule i'. We denote these as $\mathcal{R}i$.

A *path* is a sequence $\theta_1 \ldots \theta_k$ of transitions in the FA such that the source state of θ_{i+1} is the target state of θ_i, for $1 \leq i \leq k-1$. A *path through the FA* is

a path $\theta_1 \ldots \theta_k$ such that the source state of θ_1 is the start state and the target state of θ_k is an accepting state. For a path θ, we write $\overline{\theta}$ for the string of terminal and non-terminal symbols obtained by taking the labels of the transitions in θ and removing the ϵ and $\mathcal{R}$ symbols. We say that a string μ of grammar symbols is *accepted* by an FA, N, if there is a path θ through N such that $\overline{\theta} = \mu$.

3 Reduction Incorporated Automata

Parsing involves comparing a sentential form with the rules of a grammar so as to detect derivation steps and thus derivations. It is natural to render a grammar as an FA in which the states correspond to *slots* in the grammar, that is positions between grammar symbols, and the edges to the matching of grammar symbols. The standard non-deterministic LR(0) automaton is based on this approach: a slot is represented as an *item*, a rule of the form $X ::= \alpha \cdot \beta$, where $X ::= \alpha\beta$ is a grammar production rule. The automaton includes a state for each item, $X ::= \alpha \cdot x\beta$ and $X ::= \alpha x \cdot \beta$ are connected via an edge labelled x, and $X ::= \alpha \cdot A\beta$ and $A ::= \cdot\, \gamma$ are connected via an edge labelled ϵ. The accepting states of the FA are those with no out-edges, corresponding to items of the form $A ::= \gamma\cdot$. The final LR(0) automaton is then obtained by performing the subset construction.

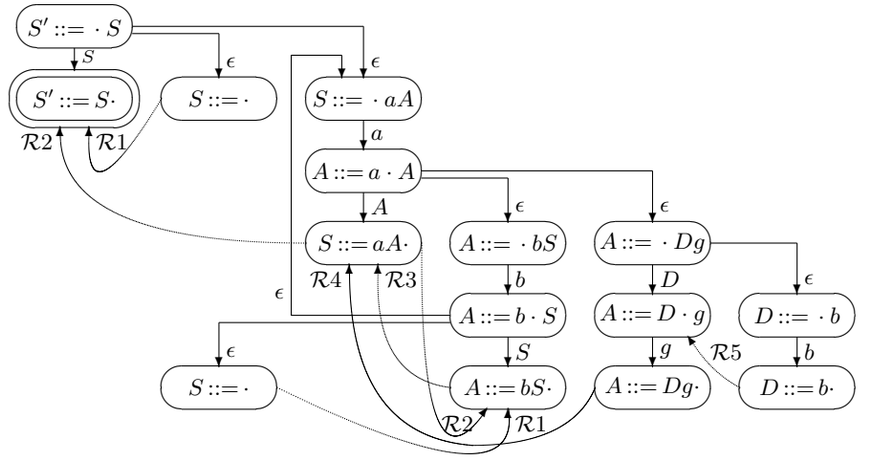
Aycock and Horspool's central idea is to add additional 'reduction' transitions from accepting states of the FA to the state which would be reached after the corresponding reduction had been performed. Precisely how these reductions should be introduced is slightly subtle and a full discussion is given in [9]. It turns out that we cannot simply use the LR(0) automaton. Aycock and Horspool give a method for constructing their FA which uses *tries* [6]. We use a different approach based on our Reduction Incorporated Automaton (RIA). The following is an informal description; a formal definition is given in Section 3.1.

(i) Create an expanded LR(0) automaton by 'multiplying out': each occurrence of a nonterminal on the RHS of a production causes the entire set of items for that non-terminal to be added afresh. In the case of recursive rules, add an ϵ-edge back to the most recent instance of the target item on a path from the start state to the current state.
(ii) Add Aycock and Horspool style 'reduction' transitions (labelled with $\mathcal{R}$) from the leaves of the multiplied out FA (which would correspond to accepting states in the LR(0) automaton) to the state corresponding to the consumption of the accepted non-terminal. The resulting automaton is called the first stage, or initial RIA (IRIA).
(iii) Remove transitions labelled with non-terminals, since the final RIA will only be used to match strings of terminals.
(iv) Perform the subset construction, with $\mathcal{R}$-transitions treated as non-ϵ edges, to remove some non-determinism.

Example 1 Given Γ_1, a right recursive grammar:

0. $S' ::= S$	1. $S ::= \epsilon$	2. $S ::= aA$
3. $A ::= bS$	4. $A ::= Dg$	5. $D ::= b$

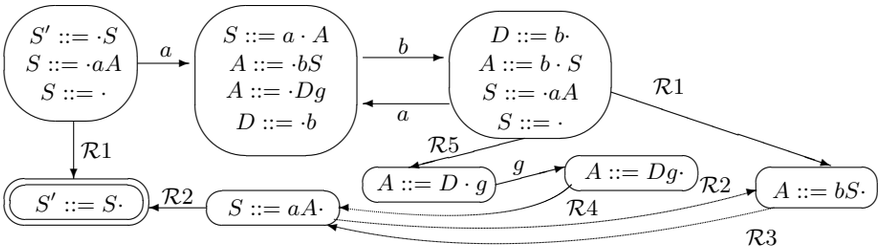
the IRIA resulting from construction steps (i) and (ii) is



In this case the difference between this and the standard LR(0) approach is that there are two states labelled $S ::= \cdot$.

The back-edge from state $A ::= b \cdot S$ to $S ::= \cdot aA$ and the corresponding $\mathcal{R}2$ transition from $S ::= aA\cdot$ to $A ::= bS\cdot$ arise from the recursive occurrence of S in the rule $A ::= bS$. These recursion back-edges indicate the points at which the 'multiplying-out' of the LR(0) automaton would yield an infinite automaton unless a cycle is created.

After application of construction steps (iii) and (iv) we obtain RIA(Γ_1):



The existence of the $\mathcal{R}$-transitions means that the automaton is still non-deterministic. We could reduce the non-determinism further by assigning 'lookahead' sets to the $\mathcal{R}$ transitions. However, this would still not always resolve all non-determinism, so we shall discuss the addition of lookahead symbols to the final push down automaton rather than introduce them at this intermediate stage.

3.1 Formal RIA Construction Algorithm

The results given in this paper (and the corresponding proofs given in [9]) are based on the following formal RIA construction algorithm.

Given an augmented grammar Γ we construct an RIA as follows:
Step 1: Create the start node labelled $S' ::= \cdot S$.

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

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

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

3. If $x = Y$, where Y is a non-terminal, for each rule $Y ::= \delta$:
   if there is a node L, labelled $Y ::= \cdot\, \delta$, and a path θ from L to K which consists of only primary edges and primary ϵ-edges (θ may be empty), add an arrow labelled ϵ from K to L,
   otherwise, create a new node with label $Y ::= \cdot\, \delta$ and add an arrow labelled ϵ from K to this new node. This arrow is defined to be a *primary ϵ-edge*.

4. Mark K as dealt with.

Step 3: Remove all the 'dealt with' marks from all nodes and mark the node labelled $S' ::= S\cdot$ as the accepting node.

Step 4: While there are nodes labelled $Y ::= \gamma\cdot$ that are not dealt with: pick a node K labelled $X ::= x_1 \ldots x_n\cdot$ which is not marked as dealt with. Let $Y ::= \gamma$ be rule i. If $X \neq S'$ then find each node L labelled $Z ::= \delta \cdot X\rho$ such that there is a path labelled $(\epsilon, x_1, \ldots, x_n)$ from L to K, then add an arrow labelled $\mathcal{R}i$ from K to the child of L labelled $Z ::= \delta X \cdot \rho$. Mark K as dealt with. (The new edge is called a reduction edge).

Step 5: Remove all arrows labelled with nonterminals (this does not make any node unreachable because there is a reduction arrow with the same target as each removed arrow).

Step 6: Perform the subset construction with edges labelled $\mathcal{R}i$ treated as non-ϵ edges.

Theorem 1 *Let Γ be an augmented grammar and let RIA(Γ) be the associated automaton constructed as above. If α is a non-trivial sentential form of Γ then α is accepted by RIA(Γ).*

Furthermore, if Γ does not contain any proper self embedding and $u \in \mathbf{T}^*$ is accepted by RIA(Γ) then $S' \stackrel{*}{\Rightarrow} u$.

In the next section we will describe how to deal with grammars which do contain proper self embedding.

4 Generalised Regular Parsing

In this section we describe how to build a PDA which can be used to recognise sentences in a language generated by a given context free grammar. The method is an extension of the construction given by Aycock and Horspool [6].

As we mentioned in the introduction, the problem with Aycock and Horspool's method is that, like any recursion based method, if a process makes a

recursive call to itself without consuming any input then the method will ultimately fail to terminate unless special terminating measures, such as limiting the number of recursive calls, are introduced. An automaton constructed by Aycock and Horspool's method can make a recursive call to itself without consuming any input if and only if the original grammar contains hidden left recursion. (Non-hidden left recursion does not generate a recursive call because it is absorbed into the appropriate state when the automaton is constructed. This is exactly analogous to LR(0) FAs which, for the same reason, admit direct left recursion but not hidden left recursion.) Elsewhere, [5], we have given an extension of Tomita's original algorithm which allows ϵ-productions and can cope with grammars which contain hidden left recursion because the algorithm checks whether a path in the GSS already exists before adding it again. Using a similar idea, we have modified the call structure used by Aycock and Horspool to allow us to determine when a call is being repeated without any input having been consumed, thus ensuring that our algorithm always terminates.

In the rest of this section we shall describe our generalised regular parsing algorithm. We begin by modifying the grammar to remove most of the recursion (which changes the language generated by the grammar). We then construct the RIA for the modified grammar as described in the previous section, together with RIAs for certain subgrammars. We describe how to construct, from these automata, an automaton, called a recursion call automaton (RCA), for the original grammar Γ which, together with a stack, can be used to recognise sentences of Γ. We then give our algorithm for computing the results of all possible traversals of the RCA for a given input string.

4.1 Recursive Call Automata

Given a grammar, Γ, if there is a non-terminal A and a derivation $A \stackrel{+}{\Rightarrow} \alpha A \beta$, where α and β are not null, then pick one such A and replace an instance of A on the RHS of a rule with a special terminal of the form $A^{\perp}$ so that this derivation is no longer possible. Repeat this process until the derived grammar, Γ_S, has no proper self embedding, and construct RIA(Γ_S).

For each special terminal $A^{\perp}$ in Γ_S construct the grammar Γ_A which has the same rules as Γ_S but with the addition of a new start rule $S_A ::= A$, and then construct RIA(Γ_A) in such a way that all the state labels are disjoint from the state labels of any other automaton we have constructed during this process. We link all these automata together as follows: for each transition anywhere in any of the automata labelled $A^{\perp}$ let the source node be labelled h and the target node be labelled k. Remove this transition from the automaton and add a new transition from node h to the start node of the automaton Γ_A, labelling the new transition $p(k)$. Label the accepting node of RIA(Γ_A) with *pop*. The start and accepting states of the RCA are the start and acceptiing states, respectively, of RIA(Γ_S). We shall refer to this new automaton as the *recursion call automaton (RCA) associated with* Γ, RCA(Γ).

We can reduce the non-determinism in RCA(Γ) by adding lookahead symbols to the transitions. All reduction and push transitions whose target has a

transition labelled $x \in \mathbf{T}$ or a transition with x in its lookahead set have x in their lookahead sets. The *pop* action has a lookahead set which is the union of all the lookahead sets associated with all the transitions from states which can be reached when this pop action is performed (i.e. the targets of the transitions labelled $A^{\perp}$ in the reduction incorporated automaton). Also, actions whose target is an accepting state have \$ in their lookahead sets.

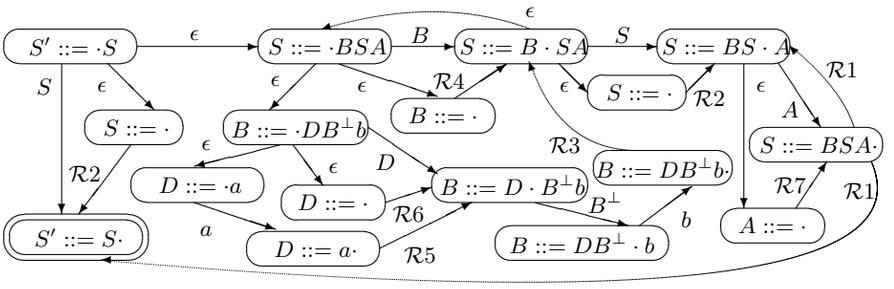
Example 2 Consider the grammar Γ which contains proper self embedding on B and hidden left and right recursion on S. (Note the string A is null so the derivation $S \Rightarrow BSA$ does not constitute proper self embedding.)

$$\begin{array}{lllllllll} S' & ::= & S & & B & ::= & DBb \mid \epsilon & & A ::= \epsilon \\ S & ::= & BSA \mid \epsilon & & D & ::= & a \mid \epsilon & & \end{array}$$

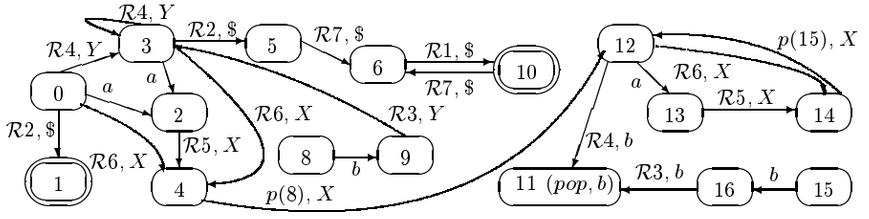
We remove the proper self embedding by replacing the second instance of B with a special terminal, $B^{\perp}$, resulting in the grammar Γ_S

$$\begin{array}{lllllllll} 0.\ S' & ::= & S & & 3.\ B & ::= & DB^{\perp}b & & 6.\ D ::= \epsilon \\ 1.\ S & ::= & BSA & & 4.\ B & ::= & \epsilon & & 7.\ A ::= \epsilon \\ 2.\ S & ::= & \epsilon & & 5.\ D & ::= & a & & \end{array}$$

The first stage reduction incorporated FA, IRIA(Γ_S) is



Building and combining RIA(Γ_S) and RIA(Γ_B) gives



(Here $X = \{a, b\}$ and $Y = \{a, b, \$\}$.)

A traversal of an RCA starts in the start state with a base node on the call stack. Then, if we have reached state h and the next input symbol is a, either,

- move to state k along a transition labelled $(\mathcal{R}i, Z)$ where $a \in Z$, or
- move to state k along a transition labelled $(p(l), Z)$ where $a \in Z$, and push l on to the top of the call stack, or
- if h is labelled (pop, Z) and $a \in Z$, pop a symbol, l, off the top of the call stack and move to state l, or
- move to state k along a transition labelled a and read the next input symbol.

If we have reached an accepting state of RCA(Γ) and all the input has been consumed, report that the input has been accepted. Otherwise, if no further transitions are possible, report that this traversal has not been successful.

Theorem 2 *A string, u, of terminals is in the language generated by Γ if and only if there is a traversal of RCA(Γ) which accepts the input u.*

4.2 Traversing an RCA

We now consider how to determine whether or not there is a traversal of RCA(Γ) on a given input string $u = a_1 \dots a_n$. The basic idea is to traverse the RCA and record the states we can reach on the input that we have seen so far. The process proceeds in steps, one step for each input symbol and one for the last symbol, \$. We start in the start state and construct the set of all states which can be reached without consuming any input. We then start a new set which contains all states which can be reached from a state in the first set along a transition labelled a_1. We then add all the states which can be reached without consuming any further input. If we encounter a transition labelled with a push action then we need to move to the state which is the target of this transition and to record the state we need to move back to, the argument of the push label, to be used when we reach the corresponding pop action. Thus we instead of just states, we maintain a set U of (state, node) pairs which can be reached from the currently read input.

The possibility of nested calls and multiple alternatives where the RCA is non-deterministic mean that we need an efficient method of recording the return states. Following Aycock and Horspool we create a call graph which is structured in a similar way to Tomita's graph structured stack and associate each state we reach in a traversal with a node in the call graph. When a *push* transition is used we find or create a node in the call graph which is labelled with the return state and we record the corresponding node with the state. We begin all traversals by creating a base node in the call graph, q_0, labelled -1 (which is not the label of any RCA node). We need to record the set of call graph nodes constructed at each step in order to check whether a node with a particular label has already been constructed, because in this case the node is reused. For this we use a set P.

We shall see that left and right recursion in Γ_S cause loops of reductions in RCA(Γ). We ensure that the traversal construction process terminates in such cases by only adding each pair (k, q) to the set U once at any given step in the process. It is also possible to have loops which consume no input if in the

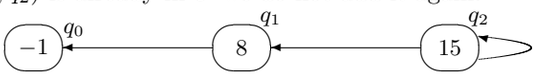
grammar Γ_S we have $A \stackrel{*}{\Rightarrow} \alpha A^{\perp} \beta$ where $\alpha \stackrel{*}{\Rightarrow} \epsilon$. In this case the loop involves a push to the start state of Γ_A and is the source of the problem with algorithm given in [6]. We deal with this problem using an idea similar to that used by Farshi in his modification of Tomita's algorithm: we introduce loops in the call graph. Before giving the formal algorithm we illustrate our approach with two examples.

Computing Traversals Using Example 2

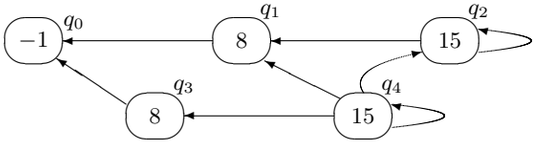
Recall the grammar from Example 2 above and consider traversing the RCA with input string $ab\$$.

We begin in the start state with lookahead symbol a and single stack node q_0, so $U = \{(0, q_0)\} = P$. From state 0 we can reach states 3 and 4 along reduction transitions, so $U = \{(0, q_0), (3, q_0), (4, q_0)\}$. From state 3 we can return to state 3 along a reduction but as $(3, q_0)$ is already in U we do not add it again, to ensure that this step terminates. State 4 has a push transition so we create a new call graph node, q_1, labelled with the argument of this transition, 8, and we make q_1 a parent of the node, q_0, and we add $(12, q_1)$ to U and P.

With lookahead symbol a from state 12 we can reach state 14 along a reduction transition, so we add $(14, q_1)$ to U. From state 14 there is a push transition $p(15)$ so we create a new call graph node q_2 labelled 15 and an edge from q_2 to q_1, and add $(12, q_2)$ to U and P. From $(12, q_2)$ we add $(14, q_2)$ to U, then from $(14, q_2)$ we traverse the push transition labelled $p(15)$. There is already a call graph node, q_2, labelled 15 so we reuse this node and create an edge from it to q_2. Since $(12, q_2)$ is already in U we do not add it again.



This step of the construction is now complete and we have $U = \{(0, q_0), (3, q_0), (4, q_0), (12, q_1), (14, q_1), (12, q_2), (14, q_2)\}$. We then read the input symbol, a, and check each of the elements in U for transitions labelled a. The states reachable in this way form the basis of the new set U for this step. Thus we have $U = \{(2, q_0), (13, q_1), (13, q_2)\}$ and $P = \emptyset$. We then traverse the reductions from 2 and 13, adding $(4, q_0)$, $(14, q_1)$ and $(14, q_2)$ to U. States 4 and 14 have push transitions, so we create new call graph nodes, q_3, q_4 labelled 8 and 15 (see the diagram below), and add $(12, q_3)$ and $(12, q_4)$ to U and P. We then traverse the transitions from state 12 and add $(14, q_3)$, $(14, q_4)$, $(11, q_3)$ and $(11, q_4)$ to U. When we process $(14, q_3)$ and $(14, q_4)$ we see that $(12, q_4) \in P$ and q_4 has label 15, so we reuse this node and just add edges from q_4 to q_3 and q_4.



State 11 has an associated *pop* action thus, when we process $(11, q_3)$, for all nodes which are children of q_3, in this case q_0, we pop the label, 8, of q_3 off the stack and move to state 8 with the new stack, whose top node is q_0, i.e. we add $(8, q_0)$ to U. Similarly, for $(11, q_4)$ we add $(15, q_4)$, $(15, q_3)$, $(15, q_2)$ and $(15, q_1)$ to U. This step is then complete with
$U = \{(2, q_0), (13, q_1), (13, q_2), (4, q_0), (14, q_1), (14, q_2), (12, q_3), (12, q_4), (14, q_3), (14, q_4), (11, q_3), (11, q_4), (8, q_0), (15, q_4), (15, q_3), (15, q_2), (15, q_1)\}$.

Finally we read the b and set $U = \{(9, q_0), (16, q_4), (16, q_3), (16, q_2), (16, q_1)\}$. With lookahead \$ there is a reduction from state 9 to 3, then to states 5, 6 and 10. So $U = \{(9, q_0), (16, q_4), (16, q_3), (16, q_2), (16, q_1), (3, q_0), (5, q_0), (6, q_0), (10, q_0)\}$. We have now completed the process, the lookahead symbol is \$ and there is an element $(10, q_0) \in U$ whose state is an accepting state of the RCA and whose call graph node is the base node. Thus we accept the input string, i.e. $ab \in L(\Gamma)$.

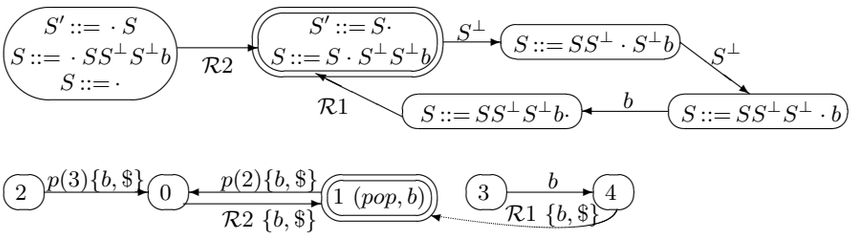
There is one further issue that we need to address before we give the formal algorithm. When we perform a pop action on a node, q labelled k, in the call graph we need to create elements in U for each of the children of q. If we subsequently add a new child p to q then we need to ensure that (k, p) is added to U. Thus when we create new edge from q to p we check to see if U contains any elements which result in a pop action from q. If such elements exist then we ensure that U contains (k, p).

Example 3

Consider the grammar Γ, $S' ::= S$ $S ::= SSSb \mid \epsilon$.
We remove the proper self embedding, resulting in the grammar Γ_S

$$0.\ S' := S \qquad 1.\ S ::= SS^{\perp}S^{\perp}b \qquad 2.\ S ::= \epsilon$$

Then RIA(Γ_S) and RCA(Γ) are

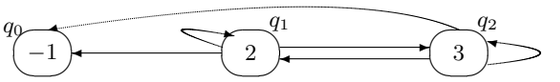


We traverse RCA(Γ) with input string $b\$$. Starting with the element $(0, q_0)$ we traverse the reduction and add the element $(1, q_0)$ to U. We then traverse the push transition, creating a new call graph node q_1 labelled 2 and adding $(0, q_1)$ to U and to P. We then traverse the reduction from $(0, q_1)$ and add $(1, q_1)$ to U. The pop action in state 1 causes $(2, q_0)$ to be added to U, and, as there is already a call graph node labelled 2 which has been constructed at this step, the push transition from $(1, q_1)$ results in the construction of an edge from q_1 to itself (see the diagram below). This creates a new edge from q_1 down which the

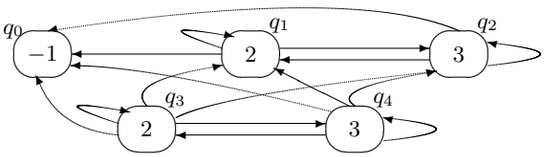
pop action associated with $(1, q_1)$ must be applied, and thus the element $(2, q_1)$ is added to U. If we then traverse the push transition from $(2, q_0)$ we create a call graph node, q_2 labelled 3 with child q_0, and add $(0, q_2)$ to U and P. When we traverse the push transition from $(2, q_1)$ we find that there is already a call graph node labelled 3 so we just add an edge from q_2 to q_1. We then traverse the reduction from $(0, q_2)$, adding $(1, q_2)$ to U, and perform the pop action for each of q_2's children, so that

$$U = \{(0, q_0), (1, q_0), (0, q_1), (1, q_1), (2, q_0), (2, q_1), (0, q_2), (1, q_2), (3, q_0), (3, q_1)\}.$$

The push transition from $(1, q_2)$ causes an edge to be added from q_1 to q_2, and then the pop action associated with $(1, q_1)$ is applied down this edge, adding $(2, q_2)$ to U. Finally, the push transition from $(2, q_2)$ causes a new edge to be added from q_2 to itself, and the pop action associated with $(1, q_2)$ then causes $(3, q_2)$ to be added to U.



This completes the first step of the process. We then read the next input symbol, b, and continue traversing the RCA. Ultimately, when this step is complete we have



$$U = \{(4, q_0), (4, q_1), (4, q_2), (1, q_0), (1, q_1), (1, q_2), (2, q_0), (2, q_1), (2, q_2), (3, q_0), (3, q_1), (3, q_2), (0, q_3), (0, q_4), (1, q_3), (1, q_4), (2, q_3), (2, q_4), (3, q_3), (3, q_4)\}.$$

Since the next input symbol is \$ and U contains $(1, q_0)$, where 1 is an accepting state of RCA(Γ), the string b is accepted.

4.3 A Formal Recognition Algorithm for RCA(Γ)

We shall now give the algorithm which computes the results of all possible traversals of an RCA for a given input. We shall assume that the RCA is given in the form of a table, $\mathcal{T}$, whose rows are indexed by the state numbers of the RCA, with the start state by convention being numbered 0, and whose columns are indexed by the terminal symbols of the grammar and the end-of-string symbol, \$. The entries in the table are sets of actions. If there is a transition in RCA(Γ) from state h to state k labelled with the terminal a, then $\mathcal{T}(h, a)$ contains the action sk. If there is a transition in RCA(Γ) from state h to state k labelled $(\mathcal{R}i, Z)$, $(p(l), Z)$ or (pop, Z) then, for all $x \in Z$, $\mathcal{T}(h, x)$ contains the action $(\mathcal{R}i, k)$, $(p(l), k)$ or *pop* respectively.

We begin with the RCA, input $a_1 \ldots a_n\$$, and a recursion call graph which contains a single node labelled -1 which is not the label of any state in the RCA. At the end of each step in the process we have a set U of RCA nodes which can be reached using the portion of input consumed so far, together with the node which is the top of the associated call stacks. In the algorithm we shall call this set U_i to facilitate the exposition. At the beginning of each step we have a set $U_{i-1}a_i$ of nodes which can be reached from the previous set via a shift action on the input symbol, a_i, which has just been read.

The nodes of the recursion call graph are all labelled with state numbers from the RCA, except for a unique base node which is labelled -1. For every node in the call graph there will be a path from this node to the base node. In practice the labels on the call graph nodes will all be states in the RCA which appear as parameters to $p()$ transitions.

```
Input: an RCA written as a table 𝒯, and a string a_1 ... a_n$
a_{n+1} = $, U_0 = P_0 = ∅, ..., U_n = P_n = ∅
create a base node, q_0, in the call graph
create a process node, u_0, in U_0 labelled (0, q_0) and add (0, q_0) to P_0
for i = 0 to n do {
  add all the elements of U_i to A
  while A ≠ ∅ {
    remove u = (h, q) from A
    if sk ∈ 𝒯(h, a_{i+1}) { if there is no node labelled (k, q) in U_{i+1} {
                             create a process node v labelled (k, q)
                             add v to U_{i+1} } }
    for each (ℛi, k) ∈ 𝒯(h, a_{i+1}) { if there is no v ∈ U_i labelled (k, q) {
                                       create a process node v labelled (k, q)
                                       add v to A and to U_i } }
    if pop ∈ 𝒯(h, a_{i+1}) { let k be the label of q and Z be the successors of q
                            for each p ∈ Z {
                              if there is no v ∈ U_i labelled (k, p) {
                                create a process node v labelled (k, p)
                                add v to A and to U_i } } }
    for each (p(l), k) ∈ 𝒯(h, a_{i+1}) {
        if there is (k, t) ∈ P_i such that t has label l {
            if there is no edge from t to q {
                add an edge from t to q
                if there is no node in U_i with label (l, q) {
                    if there is v ∈ U_i\A with label of the form (f, t)
                                  and  pop ∈ 𝒯(f, a_{i+1}) {
                        create a process node v labelled (l, q)
                        add v to A and to U_i  } } } }
        else { create a node t with label l in the call graph
               make q a successor of t
               create a process node v labelled (k, t)
               add v to A, to U_i and to P_i } } } } }
```

if U_n contains a node whose label is (h_∞, q_0) where h_∞ is an accept state
    of the RCA and q_0 is the base node of the call graph { report success }
else { report failure }

Theorem 3 *Given RCA(Γ) and an input string $a_1 \ldots a_n\$$, Algorithm 4.3 terminates and reports success if $a_1 \ldots a_n$ is in the language generated by Γ and terminates and reports failure if $a_1 \ldots a_n$ is not in the language.*

5 Conclusion

The techniques described in this paper form a two-level generalisation of the finite automata that underlie parsing. We construct an FA (the Reduction Incorporated Automaton) which recognises the regular parts of a language and use this as a basis for another FA (the Recursive Call Automaton) which uses a stack when we need to recognise recursive parts of the language.

The RCA is nondeterministic, and so we need to manage multiple stacks. Tomita's GSS is a general-purpose structure for maintaining multiple stacks which may share prefixes and, by virtue of there being a finite set of values that may appear at the top of those stacks, may also be merged. In a Tomita parser, merging occurs when two stacks have the same LR state on top. In the RCA, merging occurs when two stacks have the same RCA state on top. A key difference is that Tomita must maintain a complete trace of all stack activity because the GSS is used to manage reductions, and during a reduction the parser needs to be able to look down into the stacks to see which state to go to. This is unnecessary in the case of the RCA which only ever needs to look one level back to find the state it needs to go to.

The use of a GSS-like structure, and the applicability of our approach to generalised parsing might create the impression that our parser is Tomita-like. Indeed Aycock and Horspool whose reduction transitions inspired this work describe their algorithm as an optimisation of Tomita's algorithm. In fact, the short-circuiting of the reduction path search by the reduction transitions means that there are few points of contact between the approaches. Tomita's algorithm and the variations of it, for example those given by Farshi [4] and Rekers [10], are generalised LR algorithms in the sense that if they are given an LR grammar then they essentially behave like a traditional stack based LR parser. In our case, and that of Aycock and Horspool, the parsers behave like FAs on regular grammars.

We would like to be able to produce an efficient version of what could be called *a generalised regular parser* in the sense that the algorithm is essentially an FA in the case where the input grammar defines a regular language. However, with the method described here this cannot be fully achieved because it is possible to have a grammar which contains non-trivial self embedding but whose language is still regular, for example $S ::= aSa \mid \epsilon$. For this grammar the RIA would accept, for example, a^3. Thus, following Aycock and Horspool, our algorithm has an initial phase which modifies the input grammar if it contains proper self-embedding and

in some cases the algorithm uses a stack even though the underlying language is regular.

We have said little about the production of derivations from our parser since the scheme presented here is essentially just a recogniser. We have investigated two approaches: the production of Tomita-like Shared Packed Parse Forests and the construction of an FA whose language is essentially the set of all possible right-most derivations of the sentence. Space precludes a full discussion here, but the main issue is to ensure that production of derivations does not significantly compromise the efficiency gains from the reduction in stack activity. For this reason we prefer the second of the two approaches, see [9] for further details.

References

1. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation and Compiling. Volume 1 – Parsing of Series in Automatic Computation. Prentice-Hall Inc. (1972)
2. Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? Communications of the ACM **20** (1977)
3. Tomita, M.: Efficient parsing for natural language. Kluwer Academic Publishers, Boston (1986)
4. Nozohoor-Farshi, R.: GLR parsing for ϵ-grammars. In Tomita, M., ed.: Generalized LR parsing. Kluwer Academic Publishers, Netherlands (1991) 61–75
5. Johnstone, A., Scott, E.: Generalised reduction modified LR parsing for domain specific language prototyping. In: Proc. 35th Annual Hawaii International Conference On System Sciences (HICSS02). IEEE Computer Society, IEEE, New Jersey (2002)
6. Aycock, J., Horspool, N.: Faster generalised LR parsing. In: Compiler Construction: 8th International Conference, CC'99. Volume 1575 of Lecture Notes in computer science., Springer-Verlag (1999) 32–46
7. Aycock, J., Horspool, R.N., Janousek, J., Metichar, B.: Even faster generalised LR parsing. Acta Informatica **37** (2001) 633–651
8. Johnstone, A., Scott, E.: Tomita-style generalised LR parsers. Technical Report TR-00-12, Royal Holloway, University of London, Computer Science Department (2000)
9. Scott, E., Johnstone, A.: Table based parsers with reduced stack activity. Technical Report TR-02-08, Royal Holloway, University of London, Computer Science Department (2002)
10. Rekers, J.G.: Parser generation for interactive environments. PhD thesis, University of Amsterdam (1992)