Recognition Is Not Parsing – SPPF-Style Parsing From Cubic Recognisers

Elizabeth Scott and Adrian Johnstone

Department of Computer Science
Royal Holloway, University of London
Egham, Surrey, United Kingdom

Abstract

In their recogniser forms, the Earley and RIGLR algorithms for testing whether a string can be derived from a grammar are worst-case cubic on general context free grammars (CFG). Earley gave an outline of a method for turning his recognisers into parsers, but it turns out that this method is incorrect. Tomita's GLR parser returns a shared packed parse forest (SPPF) representation of all derivations of a given string from a given CFG but is worst-case unbounded polynomial order. The parser version of the RIGLR algorithm constructs Tomita-style SPPFs and thus is also worst-case unbounded polynomial order. We have given a modified worst-case cubic GLR algorithm, that, for any string and any CFG, returns a binarised SPPF representation of all possible derivations of a given string. In this paper we apply similar techniques to develop worst-case cubic Earley and RIGLR parsing algorithms.

Key words: Earley parsing, RIGLR parsing, cubic generalised parsing, context free languages

Since Knuth's seminal 1960's work on LR parsing [16] was extended to LALR parsers by DeRemer [6,5], the Computer Science community has been able to automatically generate parsers for a very wide class of context free languages. However, many parsers are still written manually, either using tool support or even completely by hand. This is partly because in some application areas such as natural language processing and bioinformatics we do not have the luxury of designing the language so that it is amenable to known parsing techniques, but also it is clear that left to themselves computer language designers do not naturally write LR(1) grammars. Indeed, designers sometimes prefer to write ambiguous grammars because they feel that they are easier to read.

Email address: `e.scott@rhul.ac.uk, a.johnstone@rhul.ac.uk` (Elizabeth Scott and Adrian Johnstone).

A grammar not only defines the syntax of a language, it is also the starting point for the definition of the semantics, and the grammar which facilitates semantic definition is not usually the one which is LR(1). This is illustrated by the development of the Java Standard. The first edition of the Java Language Specification [8] contains a detailed discussion of the need to modify the grammar used to define the syntax and semantics in the main part of the standard to make it LALR(1) for compiler generation purposes. In the third version of the standard [9] the compiler version of the grammar is written in EBNF and is (unnecessarily) ambiguous, illustrating the difficulty of making correct transformations. Given the difficulty in constructing natural LR(1) grammars that support desired semantics, the general parsing techniques, such as the CYK [25], Earley [7] and GLR [24] algorithms, developed for natural language processing are also of interest to the wider computer science community.

When using grammars as the starting point for semantics definition, we distinguish between *recognisers* which simply determine whether or not a given string is in the language defined by a given grammar, and *parsers* which also return some form of derivation of the string, if one exists. In their basic forms the CYK, Earley and GLR-inspired RIGLR [21] algorithms are recognisers, while standard GLR-style algorithms are designed with derivation tree construction, and hence parsing, in mind. However, in both recogniser and parser form, Tomita's GLR algorithm [24] is of unbounded polynomial order in the worst case. In this paper we describe the expansion of the Earley and RIGLR recognisers to parsers which are of worst-case cubic order.

1 Generalised parsing techniques

There is no known linear time parsing or recognition algorithm that can be used with all context free grammars. In their recogniser forms the CYK algorithm is worst-case order n^3 on grammars in Chomsky normal form and Earley's algorithm is worst-case order n^3 on general context free grammars and worst-case order n^2 on non-ambiguous grammars, where n is the length of the input string. General recognisers must, by definition, be applicable to ambiguous grammars. Expanding general recognisers to parsers raises several problems, not least because there can be exponentially many or even infinitely many derivations for a given input string.

Of course, it can be argued that ambiguous grammars reflect ambiguous semantics and thus should not be used in practice. This would be far too extreme a position to take. For example, it is well known that the if-else construct in the standard C grammar is ambiguous, but a longest match resolution results in linear time parsers that attach the 'else' to the most recent 'if', as specified by the ANSI-C semantics. The ambiguous ANSI-C standard grammar is certainly practical for parser implementation. However, in general ambiguity is not so easily handled, and it is well known that grammar ambiguity is in fact undecidable [12], thus we cannot expect a parser generator simply to check for ambiguity in the grammar and report the problem back to the user.

It is possible that many of the ad hoc methods of dealing with specific ambiguity, such as the longest match approach for if-else, can be generalised into standard classes of typical ambiguity which can be automatically tested for, see for example [4], but this remains a topic requiring further research.

Another possibility is to avoid the issue by just returning one derivation. Backtracking parsers are often implemented this way, and in [10] there is an algorithm for generating a rightmost derivation from the output of an Earley recogniser in at worst-cubic time. However, if only one derivation is returned then this creates problems for a user who wants all derivations and, even in the case where only one derivation is required, there is the issue of ensuring that it is the required derivation that is returned. Furthermore, naïve users may not even be aware that there was more than one possible derivation.

A truly general parser will return all possible derivations in some form. Perhaps the most well known representation is the shared packed parse forest (SPPF) described and used by Tomita [24]. Using this approach we can at least tell whether there is more than one derivation of a given string in a given grammar: use a GLR parser to build an SPPF and then test to see if the SPPF contains any packed nodes. Tomita's description of the representation does not allow for the infinitely many derivations which arise from grammars which contain cycles but it is relatively simple to modify his formulation to include these, and a fully general SPPF construction, based on Farshi's version [18] of Tomita's GLR algorithm, was given by Rekers [19]. These algorithms are all worst-case unbounded polynomial order and, in fact, Johnson [13] has shown that Tomita-style SPPFs are worst-case unbounded polynomial size. Thus using such structures will also turn any cubic recognition technique into a worst-case unbounded polynomial parsing technique.

The recogniser described in [2] is not applicable to grammars with hidden left recursion, but the closely related RIGLR algorithm [21] is fully general and, as we shall show, as a recogniser is of worst-case cubic order. There is a parser version which correctly constructs SPPFs but as these are Tomita-style SPPFs the parser is of unbounded polynomial order.

Leaving aside the potential increase in complexity when turning a recogniser into a parser, it is clear that this process is often difficult to carry out correctly. Earley gave an algorithm for constructing derivations of a string accepted by his recogniser, but this was subsequently shown by Tomita [24] to return spurious derivations in certain cases. In [3] there is given an outline of an algorithm to turn the recogniser reported there and in [2] into a parser, but again, as written, this algorithm will generate spurious derivations as well as the correct ones. Tomita's original version of his algorithm failed to terminate on grammars with hidden left recursion and, as remarked above, had no mechanism for constructing complete shared packed parse forests for grammars with cycles.

As we have mentioned, Tomita's GLR algorithm was designed with parse tree construction in mind. We have given a GLR algorithm, BRNGLR [23], which is worst-case cubic order and, because the tree building is integral to the algorithm, the parser, which builds a modified form of SPPF, is also worst-case cubic order. In this paper we apply similar techniques to the Earley and RIGLR recognisers and construct complete Earley and RIGLR parsers which are worst-case cubic order. In particular, we have an Earley *parser* which produces an SPPF representation of all derivations of a given input string in worst-case cubic space and time.

We begin with background material, describing derivations, shared packed parse forests and Earley's recogniser, for simplicity without lookahead. In Section 3 we discuss Earley's proposed parser and illustrate its problems. We use this to motivate our Earley parser, and, in Sections 5 and 6, give both theoretical and experimental results illustrating its run-time complexity. In Section 7 we give an overview of the RIGLR recogniser, and finally we show how the same techniques that we used for the Earley parser can be used to construct a cubic RIGLR parser.

2 Background theory

A *context free grammar* (CFG) 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 $\mathcal{P}$ of numbered grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a (possibly empty) string of terminals and non-terminals. The symbol ϵ denotes the empty string.

A *derivation step* is an element of the form $\gamma A\beta \Rightarrow \gamma\alpha\beta$ where γ and β 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 $\sigma \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$. We may also write $\sigma \overset{*}{\Rightarrow} \tau$ or $\sigma \overset{n}{\Rightarrow} \tau$ in this case.

A *sentential form* is any string α such that $S \overset{*}{\Rightarrow} \alpha$, 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 *derivation tree* is an ordered tree whose root is labelled with the start symbol, leaf nodes are labelled with a terminal or ϵ and interior nodes are labelled with a non-terminal, A say, and have a sequence of children corresponding to the symbols on the right hand side of a rule for A.

A *shared packed parse forest* (SPPF) is a representation designed to reduce the space required to represent multiple derivation trees for an ambiguous sentence. In an SPPF, nodes which have the same tree below them are shared and nodes which correspond to different derivations of the same substring from the same non-terminal are combined by creating a packed node for each *family* of children. Examples are given in Sections 3 and 4. Nodes can be packed only if their yields correspond to the same portion of the input string. Thus, to make it easier to determine whether two alternates can be packed under a given node, SPPF nodes, u, are labelled with a triple (x, j, i) where x is a grammar symbol and $a_{j+1} \dots a_i$ is the yeild of the subtree rooted at u, so $x \overset{*}{\Rightarrow} a_{j+1} \dots a_i$. To obtain a cubic algorithm we use *binarised* SPPFs which contain additional *intermediate* nodes but which are of worst-case cubic size. (The SPPF is said to be binarised because the additional nodes ensure that nodes whose children are not packed nodes have out-degree at most two.)

Earley's recognition algorithm constructs, for each position i in the input string $a_1 \dots a_n$, a set of *items*. Each item represents a position in the grammar that a top down parser could be in after matching $a_1 \dots a_i$. In detail, the set $\mathbf{E}_0$ is initially set to be the items $(S ::= \cdot\alpha, 0)$. For $i > 0$, $\mathbf{E}_i$ is initially set to be the items $(A ::= \alpha a_i \cdot \beta, j)$ such that $(A ::= \alpha \cdot a_i\beta, j) \in \mathbf{E}_{i-1}$. The sets $\mathbf{E}_i$ are constructed in order and 'completed' by adding items as follows: for each item $(B ::= \gamma \cdot D\delta, k) \in \mathbf{E}_i$ and each grammar rule $D ::= \rho$, $(D ::= \cdot\rho, i)$ is added to $\mathbf{E}_i$, and for each item $(B ::= \nu\cdot, k) \in \mathbf{E}_i$, if $(D ::= \tau \cdot B\mu, h) \in \mathbf{E}_k$ then $(D ::= \tau B \cdot \mu, h)$ is added to $\mathbf{E}_i$. The input string is in the language of the grammar if and only if there is an item $(S ::= \alpha\cdot, 0) \in \mathbf{E}_n$.

Below are the Earley sets for the grammar Γ_1, also below, and the input string aa.

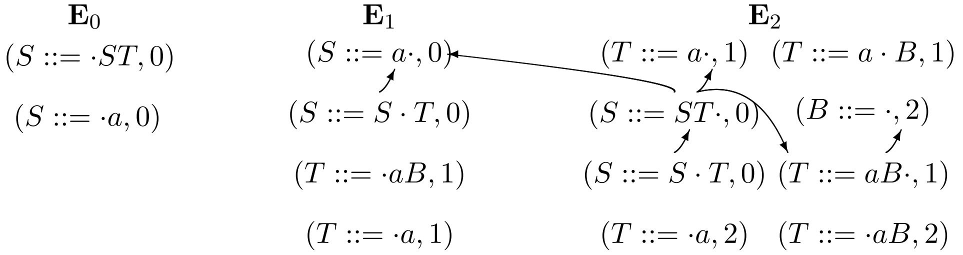
$$S ::= \; S \; T \mid a \qquad\qquad B ::= \epsilon \qquad\qquad T ::= \; a \; B \mid a$$

$$\begin{array}{ll}
\mathbf{E}_0 & = \{(S ::= \cdot ST, 0), (S ::= \cdot a, 0)\} \\
\mathbf{E}_1 & = \{(S ::= a\cdot, 0), (S ::= S \cdot T, 0), (T ::= \cdot aB, 1), (T ::= \cdot a, 1)\} \\
\mathbf{E}_2 & = \{(T ::= a \cdot B, 1), (T ::= a\cdot, 1), (B ::= \cdot, 2), (S ::= ST\cdot, 0), \\
 & \qquad (T ::= aB\cdot, 1), (S ::= S \cdot T, 0), (T ::= \cdot aB, 2), (T ::= \cdot a, 2)\}
\end{array}$$

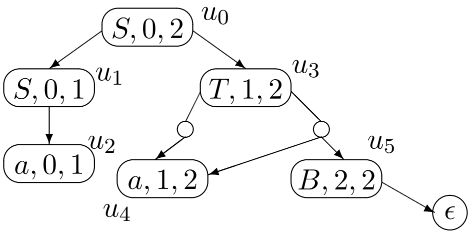
3 Problems with Earley parser construction

Earley's original paper gives a brief description of how to construct a representation of all possible derivation trees from the recognition algorithm, and claims that this requires at most cubic time and space. The proposal is to maintain pointers from the non-terminal instances on the right hand sides of a rule in an item to the item that 'generated' that item. So, if $(D ::= \tau \cdot B\mu, h) \in \mathbf{E}_k$ and $(B ::= \delta\cdot, k) \in \mathbf{E}_i$ then a pointer is assigned from the instance of B on the left of the dot in $(D ::= \tau B \cdot \mu, h) \in \mathbf{E}_i$ to the item $(B ::= \delta\cdot, k) \in \mathbf{E}_i$. In order to keep the size of the sets $\mathbf{E}_i$ in the parser version of the algorithm the same as the size in the recogniser pointers are added from the instance of B in $(D ::= \tau B \cdot \mu, h)$ to each of the items of the form $(B ::= \delta'\cdot, k')$ in $\mathbf{E}_i$.

Example 1 Applying this approach to the grammar Γ_1 from the previous section, and the string aa, gives the following structure.



From this structure the SPPF below can be constructed, as follows.



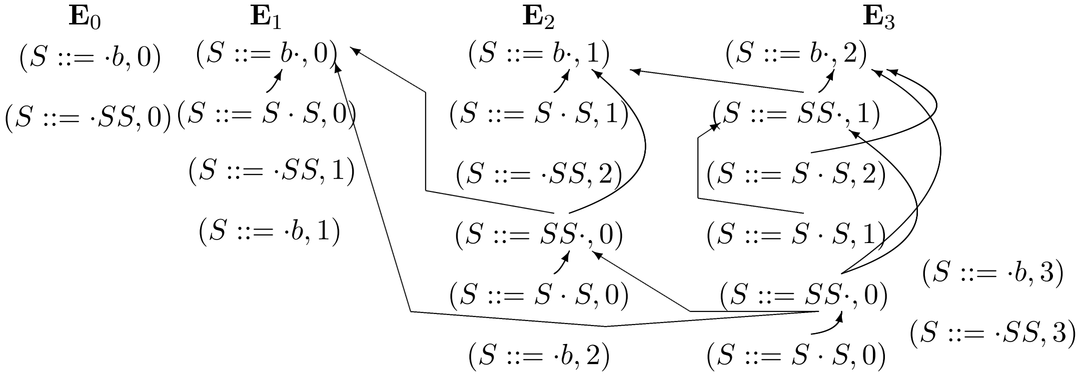
We start with $(S ::= ST\cdot, 0)$ in $\mathbf{E}_2$. Since the integer in the item is 0 and it lies in $\mathbf{E}_2$, we create a node, u_0, labelled $(S, 0, 2)$. The pointer from S points to $(S ::= a\cdot, 0)$ in $\mathbf{E}_1$, so we create a child node, u_1, labelled $(S, 0, 1)$. From u_1 we create a child node, u_2, labelled $(a, 0, 1)$. Returning to u_0, there is a pointer from T that points to $(T ::= aB\cdot, 1)$ in $\mathbf{E}_2$, so we create a child node, u_3, labelled $(T, 1, 2)$. From u_3 we create a child node u_4 labelled $(a, 1, 2)$ and, using the pointer from B, a child node, u_5, labelled $(B, 2, 2)$, which in turn has child labelled ϵ. There is another pointer from T that points to $(T ::= a\cdot, 1)$ in $\mathbf{E}_2$. We already have an SPPF node, u_3, labelled $(T, 1, 2)$ so we reuse this node. We also have a node, u_4, labelled $(a, 1, 2)$. However, u_3 does not have a family of children consisting of the single element u_4, so we pack its existing family of children and create a further packed node with child u_4.

This procedure works correctly for the above example, but adding multiple pointers to a given instance of a non-terminal can create errors.

Example 2 As remarked in [24] p74, if we consider the grammar Γ_2

$$S ::= SS \mid b$$

and the input string *bbb* we find that the above procedure constructs



which generates the correct derivations of *bbb* but also spurious derivations of the strings *bbbb* and *bb*. The problem is that the derivations of *bb* and *b* from the left-most S in the item $(S ::= SS\cdot, 0)$ of $\mathbf{E}_3$ become intertwined with the derivations of *bb* and *b* from the rightmost S.

We could avoid this problem by creating separate instances of the items for different substring matches, so if $(B ::= \delta\cdot, k), (B ::= \sigma\cdot, k') \in \mathbf{E}_i$ where $k \neq k'$ then we create two copies of $(D ::= \tau B \cdot \mu, h)$ one pointing to each item. In the above example we would create two items $(S ::= SS\cdot, 0)$ in $\mathbf{E}_3$ one in which the second S points to $(S ::= b\cdot, 2)$ and the other in which the second S points to $(S ::= SS\cdot, 1)$. This would cause correct derivations to be generated, but it also effectively embeds all the derivation trees in the construction and, as reported by Johnson, the size cannot be bounded by $O(n^p)$ for any fixed integer p. For example, using such a method for input b^n to the grammar Γ_3

$$S ::= SSS \mid SS \mid b$$

the set $\mathbf{E}_i$ constructed by the parser will contain $\Omega(i^3)$ items and hence the complete structure contains $\Omega(n^4)$ elements. Thus this version of Earley's method does not result in a cubic parser. To see this note first that, when constructed by the recogniser, the Earley set $\mathbf{E}_i$ is the union of the sets

$$\begin{aligned}
U_0 &= \{(S ::= b\cdot, i-1), (S ::= \cdot SSS, i), (S ::= \cdot SS, i), (S ::= \cdot b, i)\}\\
U_1 &= \{(S ::= S \cdot SS, k) \mid i-1 \geq k \geq 0\}\\
U_2 &= \{(S ::= S \cdot S, k) \mid i-1 \geq k \geq 0\}\\
U_3 &= \{(S ::= SS\cdot, k) \mid i-1 \geq k \geq 0\}\\
U_4 &= \{(S ::= SS \cdot S, k) \mid i-2 \geq k \geq 0\}\\
U_5 &= \{(S ::= SSS\cdot, k) \mid i-3 \geq k \geq 0\}.
\end{aligned}$$

If we add pointers then, since there are i elements $(S ::= SS\cdot, q)$ in $\mathbf{E}_i$, $0 \leq q \leq (i-1)$, and $(S ::= \cdot SSS, q) \in \mathbf{E}_q$, we will add i elements of the form

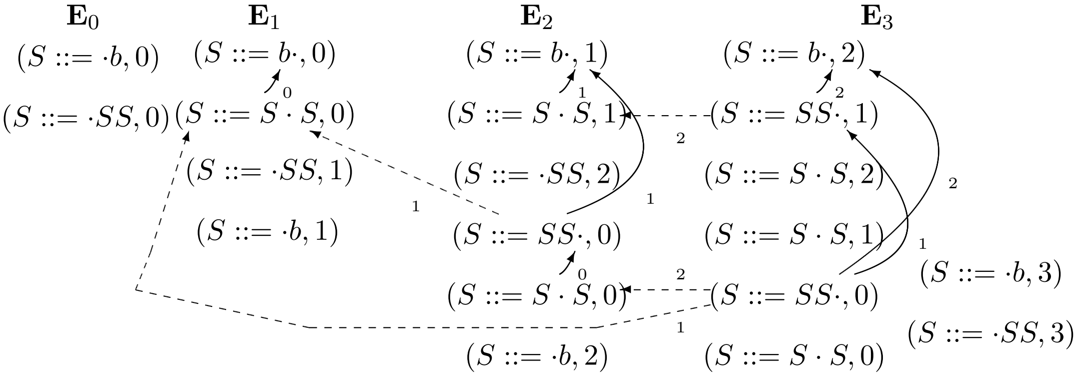
$(S ::= S{\cdot}SS, q)$ to $\mathbf{E}_i$. Then $\mathbf{E}_q$ will have q elements of the form $(S ::= S{\cdot}SS, p)$, $0 \leq p \leq (q-1)$, so we will add $i(i-1)/2$ elements of the form $(S ::= SS \cdot S, r)$ to $\mathbf{E}_i$, $0 \leq r \leq (i-1)$. Finally, $\mathbf{E}_q$ will have $q(q-1)/2$ elements of the form $(S ::= SS \cdot S, p)$, $0 \leq p \leq (q-1)$, so we will add $i(i-1)(i-3)/6$ elements of the form $(S ::= SSS\cdot, r)$ to $\mathbf{E}_i$.

Grune [11] has described a method which exploits an Unger style parser to construct the derivations of a string from the sets produced by Earley's recogniser. However, as noted by Grune, in the case where the number of derivations is exponential the resulting parser will also be of at least unbounded polynomial order in worst-case.

4 A cubic Earley parsing algorithm

We could turn Earley's algorithm into a correct parser by labelling the pointers, and allowing binarised SPPFs to be constructed by adding pointers between items rather than instances of non-terminals. We need two types of pointer, predecessor and reduction. As the $\mathbf{E}_i$ are constructed, for each item $t = (B ::= \tau\cdot, k) \in \mathbf{E}_i$, and each pair of corresponding items $q = (D ::= \tau \cdot B\mu, h) \in \mathbf{E}_k$ and $p = (D ::= \tau B \cdot \mu, h) \in \mathbf{E}_i$, add a reduction pointer labelled k from p to t and, if $\tau \neq \epsilon$, a predecessor pointer labelled k from p to q. For each $q = (A ::= \alpha \cdot a_i\beta, j) \in \mathbf{E}_{i-1}$, if $\alpha \neq \epsilon$, add a predecessor pointer labelled $i-1$ from q to p.

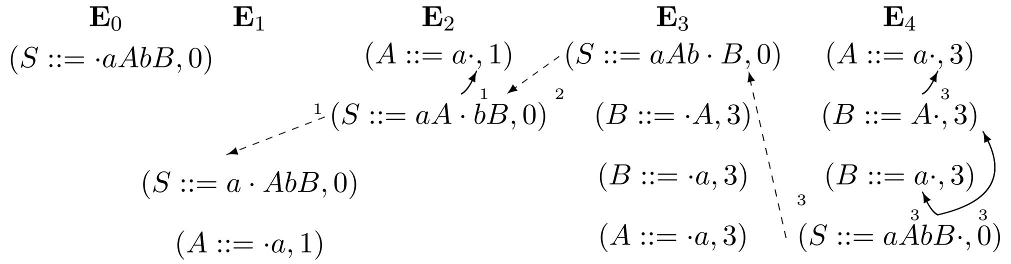
For Γ_2, above, and the string bbb we get the following structure.



(For ease of reading pointers from nodes not reachable from the node in $\mathbf{E}_3$ labelled $(S ::= SS\cdot, 0)$ have been left off the diagram).

For the grammar, Γ_4, below and the string $aaba$ we get the following structure.

$$S ::= \ a\ A\ b\ B \qquad A ::= \ a \qquad B ::= \ A \mid a$$



In [20] we present a post-parse function that walks the pointer-annotated Earley sets and outputs a binary SPPF. We shall not give this function here but it is useful to have the concept of the pointer-annotated Earley sets as it underpins the parser we now describe that constructs the SPPF as it proceeds. This parser is based on the techniques developed in [23] for constructing cubic GLR parsers, and avoids the need to actually construct the item pointers.

The SPPF constructed is similar to the binarised SPPF constructed by the BRNGLR algorithm but the additional nodes are the left hand rather than right hand children, reflecting the fact that Earley's recogniser is essentially top down rather than bottom up. An interior node, u, of the SPPF is either a symbol node labelled (B, j, i) or an intermediate node labelled $(B ::= \gamma x \cdot \delta, j, i)$. A family of children of u will consist of two nodes, or possibly just one node if u is a symbol node. If a node has more than one family of children then each family will be grouped under its own packed node.

The algorithm itself is in a form that is similar to that in which GLR algorithms are traditionally presented. There is a step in the algorithm for each element of the input string and at step i the Earley set $\mathbf{E}_i$ is constructed, along with all the SPPF nodes with labels of the form (s, j, i), $j \leq i$.

In order to construct the SPPF as the Earley sets are built, we record with an Earley item the SPPF node that corresponds to it. Thus Earley items in $\mathbf{E}_i$ are triples (s, j, w). If s is of the form $X ::= \alpha\cdot$ then $w = (X, j, i)$, if $\beta \neq \epsilon$ and s is $X ::= x \cdot \beta$ then $w = (x, j, i)$, otherwise $w = (s, j, i)$. The subtree below w will correspond to the derivation of the substring $a_{j+1} \dots a_i$. Earley items of the form $(A ::= \cdot\beta, j)$ do not have associated SPPF nodes, so we use the dummy node *null* in this case.

The items in each $\mathbf{E}_i$ have to be 'processed' either to add more elements to $\mathbf{E}_i$ or to form the basis of the next set $\mathbf{E}_{i+1}$. Thus when an item is added to $\mathbf{E}_i$ it is also added to a set $\mathcal{Q}$, if it is of the form $(A ::= \alpha \cdot a_{i+1}\beta, h, w)$, or to a set $\mathcal{R}$ otherwise. Elements are removed from $\mathcal{R}$ as they are processed and when $\mathcal{R}$ is empty the items in $\mathcal{Q}$ are processed to initialise $\mathbf{E}_{i+1}$.

There is a special case when an item of the form $(A ::= \alpha\cdot, i, w)$ is in $\mathbf{E}_i$, this happens if $A \Rightarrow \alpha \overset{*}{\Rightarrow} \epsilon$. When this item is processed items of the form $(X ::= \tau \cdot A\delta, i, v) \in \mathbf{E}_i$ have to be considered and it is possible that such an item may be created after the item $(A ::= \alpha\cdot, i, w)$ has been processed. Thus we use a set $\mathcal{H}$ and, when $(A ::= \alpha\cdot, i, w)$ is processed, the pair $(A, (A, i, i))$ is added to $\mathcal{H}$. Then when $(X ::= \tau \cdot A\delta, i, v)$ is processed elements of $\mathcal{H}$ are checked and appropriate action is taken.

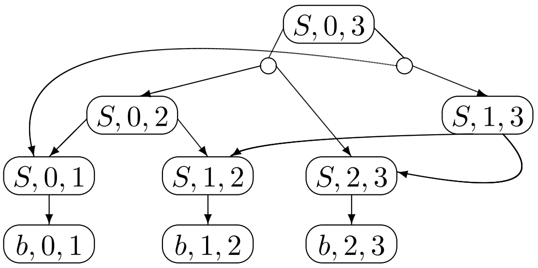
When an SPPF node is needed we first check to see if one with the required label already exists. To facilitate this checking the SPPF nodes constructed at the current step are added to a set $\mathcal{V}$.

In the following algorithm Σ_N denotes the set of all strings of terminals and non-terminals that begin with a non-terminal, together with the empty string.

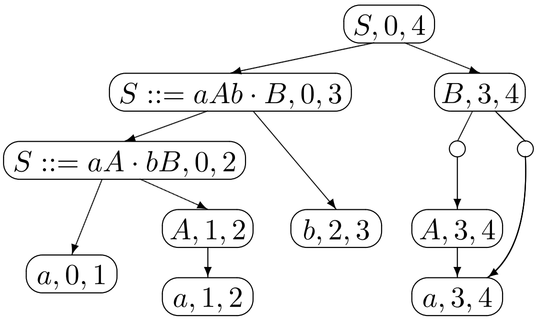
Input: a grammar $\Gamma = (\mathbf{N}, \mathbf{T}, S, \mathcal{P})$ and a string $a_1 a_2 \ldots a_n$

```
EARLEY_PARSER {

$\mathbf{E}_0, \ldots, \mathbf{E}_n$, $\mathcal{R}$, $\mathcal{Q}'$, $V = \emptyset$
for all $(S ::= \alpha) \in \mathcal{P}$ {  if $\alpha \in \Sigma_N$ add $(S ::= \cdot\alpha, 0, null)$ to $\mathbf{E}_0$
                                if $\alpha = a_1\alpha'$ add $(S ::= \cdot\alpha, 0, null)$ to $\mathcal{Q}'$ }
for $0 \leq i \leq n$ {
  $\mathcal{H} = \emptyset$,  $\mathcal{R} = \mathbf{E}_i$, $\mathcal{Q} = \mathcal{Q}'$
  $\mathcal{Q}' = \emptyset$
  while $\mathcal{R} \neq \emptyset$ {
    remove an element, $\Lambda$ say, from $\mathcal{R}$
    if $\Lambda = (B ::= \alpha \cdot C\beta, h, w)$ {
      for all $(C ::= \delta) \in \mathcal{P}$ {
        if $\delta \in \Sigma_N$ and $(C ::= \cdot\delta, i, null) \notin \mathbf{E}_i$ {
                    add $(C ::= \cdot\delta, i, null)$ to $\mathbf{E}_i$ and $\mathcal{R}$ }
        if $\delta = a_{i+1}\delta'$ { add $(C ::= \cdot\delta, i, null)$ to $\mathcal{Q}$ } }
      if $((C, v) \in \mathcal{H})$ {
        let $y = MAKE\_NODE(B ::= \alpha C \cdot \beta, h, i, w, v, \mathcal{V})$
        if $\beta \in \Sigma_N$ and $(B ::= \alpha C \cdot \beta, h, y) \notin \mathbf{E}_i$ {
                    add $(B ::= \alpha C \cdot \beta, h, y)$ to $\mathbf{E}_i$ and $\mathcal{R}$ }
        if $\beta = a_{i+1}\beta'$ { add $(B ::= \alpha C \cdot \beta, h, y)$ to $\mathcal{Q}$ } } }
    if $\Lambda = (D ::= \alpha\cdot, h, w)$ {
     if $w = null$ {
        if there is no node $v \in \mathcal{V}$ labelled $(D, i, i)$ create one
        set $w = v$
        if $w$ does not have family $(\epsilon)$ add one }
      if $h = i$ { add $(D, w)$ to $\mathcal{H}$ }
      for all $(A ::= \tau \cdot D\delta, k, z)$ in $\mathbf{E}_h$ {
        let $y = MAKE\_NODE(A ::= \tau D \cdot \delta, k, i, z, w, \mathcal{V})$
        if $\delta \in \Sigma_{\mathbf{N}}$ and $(A ::= \tau D \cdot \delta, k, y) \notin \mathbf{E}_i$ {
                   add $(A ::= \tau D \cdot \delta, k, y)$ to $\mathbf{E}_i$ and $\mathcal{R}$ }
        if $\delta = a_{i+1}\delta'$ { add $(A ::= \tau D \cdot \delta, k, y)$ to $\mathcal{Q}$ } } }
  }
  $\mathcal{V} = \emptyset$
  create an SPPF node $v$ labelled $(a_{i+1}, i, i+1)$
  while $\mathcal{Q} \neq \emptyset$ {
     remove an element, $\Lambda = (B ::= \alpha \cdot a_{i+1}\beta, h, w)$ say, from $\mathcal{Q}$
     let $y = MAKE\_NODE(B ::= \alpha a_{i+1} \cdot \beta, h, i+1, w, v, \mathcal{V})$
     if $\beta \in \Sigma_{\mathbf{N}}$ { add $(B ::= \alpha a_{i+1} \cdot \beta, h, y)$ to $\mathbf{E}_{i+1}$ }
     if $\beta = a_{i+2}\beta'$ { add $(B ::= \alpha a_{i+1} \cdot \beta, h, y)$ to $\mathcal{Q}'$ }
  }
}
if $(S ::= \tau\cdot, 0, w) \in \mathbf{E}_n$ return $w$
else return  FAILURE
}

$MAKE\_NODE(B ::= \alpha x \cdot \beta, j, i, w, v, \mathcal{V})$ {
   if $\beta = \epsilon$ { let $s = B$ }  else { let $s = (B ::= \alpha x \cdot \beta)$ }
   if $\alpha = \epsilon$ and $\beta \neq \epsilon$ { let $y = v$ }
   else {
     if there is no node $y \in \mathcal{V}$ labelled $(s, j, i)$ create one and add it to $\mathcal{V}$
     if $w = null$ and $y$ does not have a family of children $(v)$ add one
     if $w \neq null$ and $y$ does not have a family of children $(w, v)$ add one }
   return $y$
}
```

Example 4 For Γ_2 and input *bbb* the Earley parser constructs the SPPF



For Γ_3 and input *aaba* the Earley parser constructs the SPPF



5 The order of the parser

(As we have done throughout the paper, in this section we use n to denote the length of the input to the parser.)

A formal proof that the binarised SPPFs constructed by the BRNGLR algorithm contain at most $O(n^3)$ nodes and at most $O(n^3)$ edges is given in [23]. The proof that the binarised SPPFs constructed by the parsers described in this paper are of at most cubic size is the same, and we do not give it here. Intuitively however, the non-packed nodes are characterised by an LR(0)-item and two integers, $0 \leq j \leq i \leq n$, and thus there are at most $O(n^2)$ of them. Packed nodes are children of some non-packed node, labelled (s, j, i) say, and for a given non-packed node the packed node children are characterised by an LR(0)-item and an integer l which lies between j and i. Thus each non-packed node has at most $O(n)$ packed node children and there are at most $O(n^3)$ packed nodes in a binarised SPPF. As non-packed nodes are the source of at most $O(n)$ edges and packed nodes are the source of at most two edges, there are also at most $O(n^3)$ edges in a binarised SPPF.

For the Earley parser given in Section 4, the while-loop that processes the elements in $\mathcal{R}$ executes once for each element added to $\mathbf{E}_i$. For each triple (s, j, i) there is at most one SPPF node labelled with this triple, and thus there are at most $O(n)$ items in $\mathbf{E}_i$. So the while-loop executes at most $O(n)$ times. As we have already remarked, it is possible to implement the SPPF to allow n-independent look-up time for a given node and family of children. Thus, within the while-loop for $\mathcal{R}$, the only case that triggers potentially n-dependent behaviour is the case when the item chosen is of the form $(D ::= \alpha\cdot, h, w)$. In this case the set $\mathbf{E}_h$ must be searched. This is a worst-case $O(n)$ operation. The while-loop that processes $\mathcal{Q}$ is not n-dependent, thus the integrated parser is worst-case $O(n^3)$.

6 Experimental results

In this section we (i) look at the practicality of Earley parsing for programming language style grammars, illustrating the relative costs of the parser and recogniser versions of the algorithm and (ii) compare the performance of our cubic Earley parser against the BRNGLR cubic GLR parsing algorithm. For (ii) we add lookahead to our Earley parser.

We have described elsewhere our GTB tool [14] which provides a framework for implementing and experimenting on parsing algorithms. The cubic Earley parser described in Section 4, and a recogniser-only variant, have been added to GTB, and experiments performed using the following grammars and test strings.

- The ANSI C standard grammar and the source code for (a) a Quine McCluskey minimiser (4,291 tokens) and (b) the source code for an earlier version of GTB (36,827 tokens).
- The ISO PASCAL grammar and the source code for a tree visualisation application (4,425 tokens).
- The example grammar Γ_2 for which Earley's own proposed parser is incorrect.
- The example grammar Γ_3 which has $O(n^4)$ derivation trees for strings of the form b^n.

In all cases, the input is pre-tokenised and held in memory as an array of unsigned integers; thus the parse-time overhead of fetching lexemes is negligible.

Timing data was generated using an Intel Core Duo T7300 processor clocked at 2GHz with 2GByte of physical memory running MS Windows Vista. We do not make strong claims for the absolute performance of this implementation. In particular, the data structures used to implement the SPPF and Earley sets are drawn from GTB's prototyping library: these are engineered to support debugging and tracing and we typically see improvements by factors of 5-12 when GTB's prototype algorithms are re-implemented for performance.

6.1 The practicality and relative costs of Earley recognition and parsing

Although our Earley parser is worst-case $O(n^3)$, in general it will be slower than the corresponding recogniser. Both the recogniser and the parser are worst-case cubic, but the cardinality of the recogniser Earley sets is quadratic, whereas the SPPF is worst-case cubic in size. The time taken to allocate SPPF nodes is therefore worst-case cubic, and this impacts the constants of proportionality.

In principle, we might imagine that the cardinality of the parser Earley sets would be higher than that of the recogniser, because Earley parser items include SPPF nodes. In fact each recogniser item corresponds to a unique SPPF node, which is only added to the parser item so that it can be found without searching.

Some care is required when implementing the algorithm because of the need to search the SPPF for a particular node, and the need to check a node to see if it has a particular family. Our implementation performs both of these steps in unit time.

Table 1
Earley recogniser and parser comparison

Grammar	Input	$\lvert\mathbf{E}\rvert$	Recogniser/CPU s	Parser/CPU s
C	4,291	279,601	0.14	0.19
C	36,827	2,422,300	1.23	1.93
Pascal	4,425	133,076	0.11	0.14
Γ_2	b^{300}	90,902	3.43	12.79
Γ_3	b^{200}	100,504	3.59	10.70

We implemented unit time lookup for SPPF nodes and families by exploiting the fact that a family of (s, j, i) is determined by an item $t = (X ::= \alpha x \cdot \beta)$ and the left index, k, of its right child, (x, k, i). (So s will be either t or X.) We create a two dimensional array, indexed by j and s, whose entries are triples (L, N, K). When an SPPF node $w = (s, j, i)$ node is created, the corresponding cell's L-field is set to i, and the N-field is set to point to w. When a second family is added to w, a vector of possible k values is created, and pointed to by the cell's K-field. As each subsequent family is created, the vector's k-th element is set to i.

Table 1 reports the total number, $\lvert\mathbf{E}\rvert$, of Earley items constructed and the run-time in CPU seconds, for each test case. The numbers of Earley items for Γ_2 and Γ_3 should be compared with the number of packed nodes shown in Table 2.

Compiler writers and users of traditional parser generators want to know how the new generalised techniques compare with their favourite deterministic technique. Scott McPeak [17] has reported on the performance of the C++ parser in versions 2.95.3 and 3.3.2 of gcc processing a selection of modules from the Mozilla 1.0 source code on an unspecified processor. He found that gcc 2.95.2 parsed between 43.4 and 158.9 thousand lines per second (kLPS) of pre-processed input, and gcc 3.3.2 between 22.8 and 119.2 kLPS. These times are clearly not critical within the overall time taken to compile and link applications: the parser's performance drops by as much as 55% when moving from version 2 to 3 which was presumably acceptable to the implementers. This is not a very formal analysis, but we can perhaps conclude that parse times of around 25kLPS are adequate for compiler front ends.

Our prototype implementations achieve speeds of between 5.6 and 5.7kLPS for recognition and 3.5-4.2 kLPS for parsing. Through hand-crafted data structures and control flow, it should be possible to make significant speed improvements that result in performance comparable to gcc's parser for C."

Some caveats are required: McPeak's figures quote lines of pre-processed input: we have assumed that whitespace and comment lines are suppressed by the preprocessor. More significantly, a better measure of parser performance is the total number of tokens being processed per second, since different programming styles yield different numbers of token per line.

Of course, deterministic and general parsers are incommensurate in that deterministic parsers will not admit many interesting grammars, so performance comparisons are rather artificial. Nevertheless, we believe that were practical general parsers available in the 1970's, programming languages would be more comfortable to use and note that some Eiffel compilers do indeed use conventional Earley parsers.

6.2 Cubic lookahead Earley parsing and BRNGLR parsing

We have previously reported some experimental results comparing the recogniser versions of BRNGLR and Earley's algorithm [15]. Here we present results comparing the parser versions of these algorithms.

The BRNGLR algorithm can be run on LR(0), SLR(1) or LR(1) tables, while the Earley parser described above is essentially not using any lookahead. Thus we have implemented both the algorithm given in Section 4 and a version, Earley(1), that uses a form of lookahead that essentially corresponds to an SLR(1) parser. We define

$$SL(B ::= \alpha \cdot \beta) = \begin{cases} (\textsc{first}(\beta)\backslash\{\epsilon\}) \cup \textsc{follow}(B) & \text{if } \beta \overset{*}{\Rightarrow} \epsilon \\ \textsc{first}(\beta) & \text{otherwise} \end{cases}$$

where $\textsc{first}(\beta)$ and $\textsc{follow}(B)$ are the standard sets as defined, for example, in [1]. Earley(1) checks the current input symbol against $SL(B ::= \alpha{\cdot}\beta)$ before creating an item of the form $(B ::= \alpha \cdot \beta, h, w)$. Correspondingly, we used BRNGLR running on SLR(1) tables. We report, in Table 2, the numbers of symbol, intermediate and packed SPPF nodes (s-nodes, i-nodes and p-nodes respectively) and the run-time in CPU seconds.

The numbers of symbol nodes in the Earley(1) SPPF for ϵ-grammars can be higher than those in the corresponding BRNGLR SPPF because the latter handles right nullable rules, rules of the form $X ::= \alpha\beta$ where $\beta \overset{*}{\Rightarrow} \epsilon$, in a special way. A library of SPPFs for the derivations $\beta \overset{*}{\Rightarrow} \epsilon$ is built with the compiler, and when the parser encounters a right nullable rule it truncates its search and simply inserts the preconstructed tree from the library. For example, for the grammar $S ::= SaB \mid a \quad B ::= b \mid \epsilon$ and the string a^7, the Earley(1) SPPF will contain six nodes labelled (B, i, i), $2 \leq i \leq 7$, whereas the BRNGLR SPPF will contain one node labelled B.

Table 2
Earley, Earley(1) and BRNGLR SPPF comparison

		Input	s-nodes	i-nodes	p-nodes	secs
Earley	C	4,291	38,733	4,126	82	0.19
Earley(1)	C	4,291	27,997	2,312	58	0.12
BRNGLR	C	4,291	27,997	2,062	58	0.11
Earley	C	36,827	349,518	38,870	1,101	1.91
Earley(1)	C	36,827	257,012	21,234	674	1.23
BRNGLR	C	36,827	257,012	20,289	674	1.45
Earley	Pascal	4,425	31,015	4,830	6	0.14
Earley(1)	Pascal	4,425	21,258	2,690	2	0.11
BRNGLR	Pascal	4,425	16,983	2,568	2	0.07
Earley	Γ_2	b^{300}	45,150	0	4,499,651	12.79
Earley(1)	Γ_2	b^{300}	45,150	0	4,499,651	13.03
BRNGLR	Γ_2	b^{300}	45,150	0	4,499,651	10.09
Earley	Γ_3	b^{200}	20,300	19,900	3,979,602	10.70
Earley(1)	Γ_3	b^{200}	20,300	19,701	3,959,703	10.98
BRNGLR	Γ_3	b^{200}	20,300	19,701	3,959,703	17.06

The Earley parser can also construct more intermediate nodes than the corresponding BRNGLR parser. This is because it is a top-down parser and it it creates an intermediate node when a left hand portion of a rule has been matched. BRNGLR parsers are bottom-up and only create the intermediate nodes when the whole rule has been matched and a reduction is performed. For example, for the grammar $S ::= Aaab \mid aaac \quad A ::= a$ and the string $aaab$, the Earley parser will create an intermediate node labelled $(S ::= aa \cdot ac, 0, 2)$ but the BRNGLR parser will not. Of course, in all cases nodes that are not reachable from the start node can be removed by a single pass, post parse SPPF tree walk.

7 The RIGLR recogniser

RIGLR parsers [21] are based on an approach originated in [2], whose aim was to improve the efficiency of GLR parsers by reducing parse stack activity.

LR parsers traverse an underlying DFA, recording the path taken on an associated stack. When an accepting (reduction) state is reached the path taken is effectively retraced, by popping states off the stack which correspond to the right hand side of the reduction, and the traversal continues from the state reached. For the RIGLR algorithm, the underlying DFA is expanded out, so that different instances of non-terminals on the right hand side of grammar rules generate different DFA states, and then explicit reduction transitions are added from each accepting state to the state from which the traversal should continue. This avoids the need to record the path taken on a stack. However, reduction transitions cannot be added at places in the DFA which correspond to recursion in the grammar. Thus at these points a call is made to a sub-DFA and a recursion call stack is used to manage these calls. In general, the automaton traversal will not be deterministic. The RIGLR algorithm takes a GLR approach in that at points of nondeterminism all possible actions are pursued in parallel, and the corresponding call stacks are combined into a Tomita-style graph structured stack (GSS).

In this section we give an overview of the RIGLR recognition technique and show that it has worst-case cubic order. In Section 8 we shall show how to modify the algorithm to construct binary SPPFs.

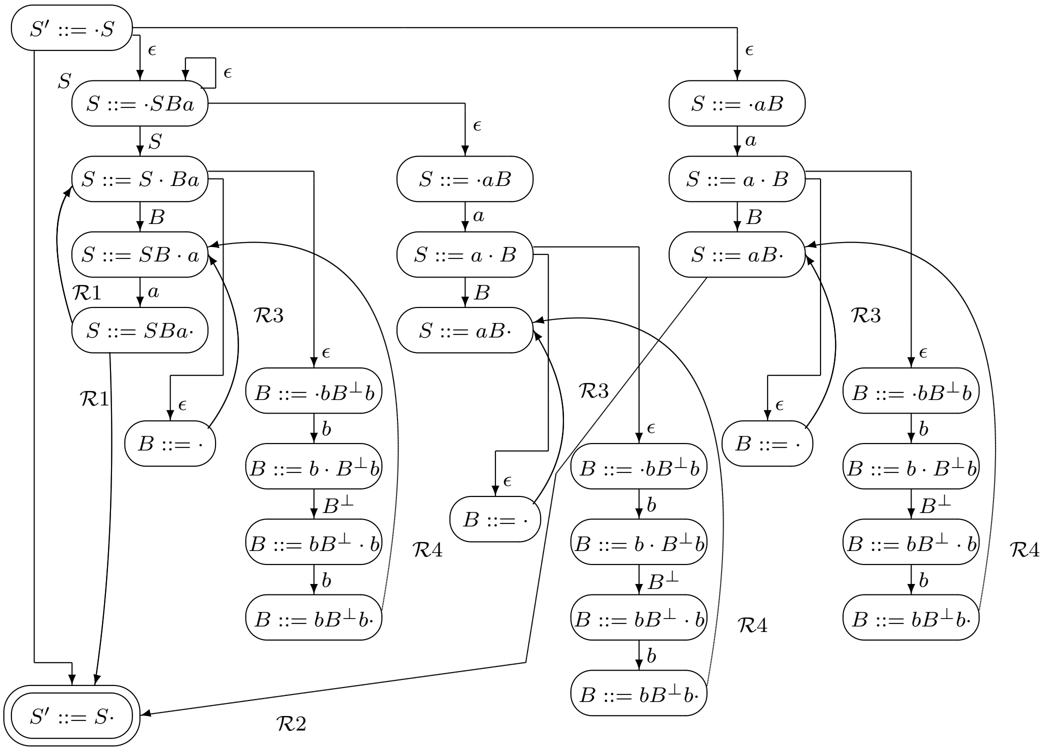
The RIGLR algorithm takes as input a push down automaton (PDA), RCA(Γ) and a string u and determines whether or not the string is in the language of the PDA. A formal definition of RCA(Γ) is given in [21]. Informally, RCA(Γ) is defined as follows. We begin by augmenting Γ with a new start rule $S' ::= S$, if it is not already augmented. Next we *terminalise* the grammar as follows. If there exists a nonterminal, Y say, such that $Y \overset{+}{\Rightarrow} \alpha Y \beta$ or such that $Y \overset{+}{\Rightarrow} \alpha Y \overset{+}{\Rightarrow} Y$, where $\alpha, \beta \neq \epsilon$, then choose such a derivation and replace an instance of Y on the right hand side of a grammar rule with a special terminal of the form $Y^\perp$, so that the derivation is no longer possible. Continue modifying the grammar in this way until there are no such derivations. We shall refer to the new grammar as Γ_S.

For example we use the following grammar, Γ_5, and its terminalisation

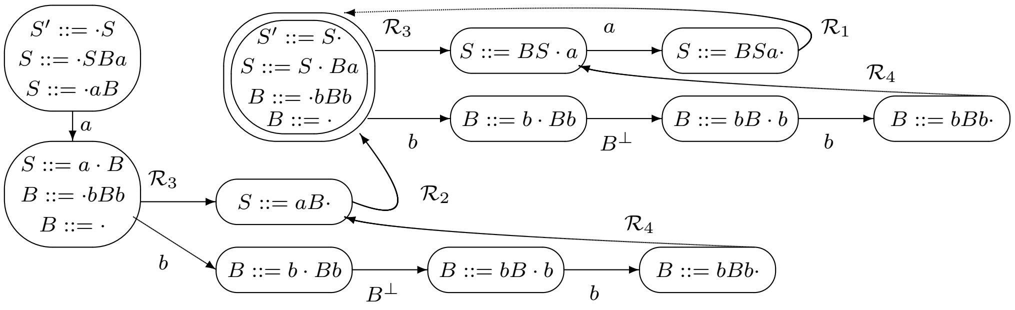
1. $S ::= S\ B\ a$ 2. $S ::= a\ B$ 3. $B ::= \epsilon$ 4. $B ::= b\ B\ b$

1. $S ::= S\ B\ a$ 2. $S ::= a\ B$ 3. $B ::= \epsilon$ 4. $B ::= b\ B^\perp\ b$

We construct a finite state automaton IRIA(Γ) whose states are labelled with *items* of the form $X ::= \alpha \cdot \beta$. The start state is labelled with the item $S' ::= \cdot S$. The remaining states are constructed recursively as follows. A state with a label of the form $X ::= \alpha \cdot a\beta$, where a is a terminal, has a transition labelled a to a new state labelled $X ::= \alpha a \cdot \beta$. For each grammar rule $A ::= \gamma$, a state h with a label of the form $X ::= \alpha \cdot A\beta$ has a transition labelled ϵ to a new state labelled $A ::= \cdot\gamma$ unless h already has an ancestor k with this label, in which case it has an ϵ-labelled transition to k. For a state h with label $A ::= \gamma\cdot$, where $A ::= \gamma$ is rule i, suppose that g is the closest ancestor of h labelled $A ::= \cdot\gamma$ and that k is the sibling of g with a label of the form $X ::= \alpha A \cdot \beta$. Then we add a transition labelled $\mathcal{R}_i$ from h to k. The accepting state of IRIA(Γ_S) is the state labelled $S' ::= S\cdot$.

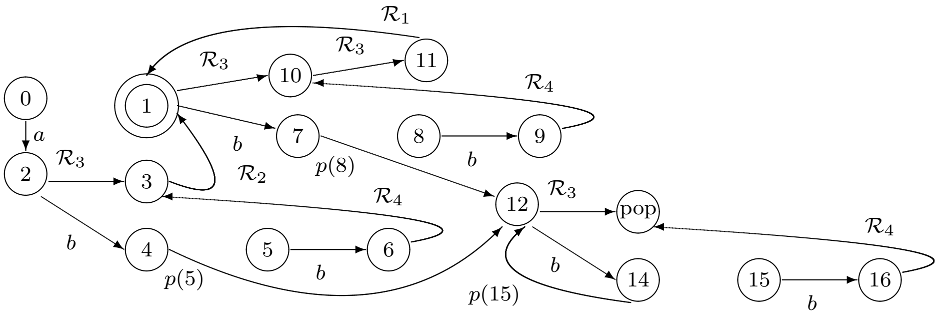


We then form RIA(Γ_S) from IRIA(Γ) by removing transitions labelled with nonterminals and then running the standard subset construction. (The states of RIA(Γ) are labelled with the items from the component IRIAstates, with the terminalisation notation removed.)



For each nonterminal $Y \neq S$ with a terminalisation we create a grammar Γ_Y from Γ_S by adding a new start rule $S_Y ::= Y$, and construct RIA(Γ_Y).

Finally we create a PDA, RCA(Γ), from the automata by replacing transitions labelled $Y^{\perp}$ with a push action that pushes the target of the transition onto the stack and goes to the start state of RIA(Γ_Y). The pop states of the PDA are the accepting states of the RIA(Γ_Y), where Y has a terminalisation, and the accepting states are the accepting states of RIA(Γ_S). Below is RCA(Γ_5).



The RCAis represented as a table $\mathcal{T}(\Gamma)$ whose rows are labelled with the states and whose columns are labelled with the terminals of Γ and the special end-of-string symbol \$. An entry $\mathcal{T}(\Gamma)(l, a)$ in the table is a finite set of *actions*. The actions are each of the form sh, $p(k, h)$, $\mathcal{R}(i, h)$ or *pop*, where h and k are states and i is a grammar rule number. The first row of $\mathcal{T}(\Gamma)$ is labelled with 0, the start state, and if l is an accepting state then *acc* is added to $\mathcal{T}(\Gamma)(l, \$)$. We traverse $\mathcal{T}(\Gamma)$ with a given input string, consuming input symbols and pushing and popping elements from the stack in the usual fashion for a push down automaton. A string u is *accepted* by RCA(Γ) if there is a traversal of the table that reads all the input and terminates in an accepting state with an empty stack. The following result is proved in [21].

Theorem 1 *A string, u, of terminals is in the language generated by Γ if and only if u is accepted by RCA(Γ).*

RIGLR recognition algorithm

In order to use Theorem 1 to determine whether a given string u is a sentence in Γ we need an algorithm which determines whether or not there is an execution path through $\mathcal{T}(\Gamma)$ on u. It is proved in [21] that the following algorithm terminates and reports success if $u \in L(\Gamma)$ and terminates and reports failure otherwise.

The algorithm operates by maintaining at step i a set U_i of *descriptors* of the form (h, q). The descriptor (h, q) lies in U_i if there is traversal on $a_1 \dots a_i$ that ends at state h with q on the top of the stack. The potential number of stacks is exponential, so these stacks are combined into a Tomita style graph structured stack, called the recursion call graph (RCG).

A set $\mathcal{A}$ is used to hold the descriptors, for the current step, that have not yet been processed, and sets P_i hold the RCG nodes constructed at step i and a flag to indicate if the node has been popped. This is required so that if additional edges are created from nodes that have already been popped then the pop action can be retrospectively applied down the new edge.

```
input: an RCA written as a table $T$, and a string $a_1 \dots a_n$
define $a_{n+1} = \$$, and set $U_0, \dots, U_n$ and $P_0, \dots, P_n$ to $\emptyset$
create a base node, $q_0$, in the RCG and add $(q_0, 0)$ to $P_0$
create an element, $u_0 = (0, q_0)$, in $U_0$
for $i = 0$ to $n$ do {
  add all the elements of $U_i$ to $\mathcal{A}$
  while $\mathcal{A} \neq \emptyset$ {
    remove $u = (h, q)$ from $\mathcal{A}$
    if $sk \in T(h, a_{i+1})$ { if $(k, q) \notin U_{i+1}$ add $(k, q)$ to $U_{i+1}$ }
    for each $\mathcal{R}(j, k) \in T(h, a_{i+1})$ { if $(k, q) \notin U_i$ add $(k, q)$ to $\mathcal{A}$ and $U_i$ }
    if $pop \in T(h, a_{i+1})$ {
          let $k$ be the label of $q$ and $Z$ be the set of successors of $q$
          if $(q, 0) \in P_i$ remove $(q, 0)$ and add $(q, 1)$ to $P_i$
          for each $p \in Z$ { if $(k, p) \notin U_i$ add $(k, p)$ to $\mathcal{A}$ and $U_i$ } }
    for each $p(l, k) \in T(h, a_{i+1})$ {
        if there is $(t, F) \in P_i$ such that $t$ has label $l$ {
              if there is no edge from $t$ to $q$ {
                 add an edge from $t$ to $q$
                 if $F = 1$ { if $(l, q) \notin U_i$ { add $(l, q)$ to $\mathcal{A}$ and to $U_i$ } } } }
          else { create a node $t$ with label $l$
                make $q$ a successor of $t$
                add $(k, t)$ to $\mathcal{A}$ and $U_i$ and add $(t, 0)$ to $P_i$ } } } }
if $U_n$ contains $(l, q_0)$ where $l$ is an accept state of the RCA { report success }
else { report failure }
```

Theorem 2 *The RIGLR recognition algorithm is worst-case cubic.*

Proof Let M be the number of rows in $\mathcal{T}(\Gamma)$. The only place where nodes in the RCG are created is when processing an action $p(l, k)$, and a new node is only created if one labelled l has not already been created at this step. Thus there are at most $(i+1)M$ nodes and $(i+2)(i+1)M^2/2$ edges in the RCG at step i, and U_i contains at most $(i+1)M^2$ elements. So the **while** loop is executed at most $O(i)$ times at Step i of the algorithm.

The number of actions $\mathcal{R}(j, k)$ in an entry $\mathcal{T}(\Gamma)(h, a_{i+1})$ is bounded by the number of non-terminal instances in the grammar rules, as is the number of actions $p(l, k)$. Thus, provided that the data structures are implemented so that sets can have constant look-up time (this can be done, see Section 6.1, as the sets are of size at most $O(n^2)$), the only part of the **while** loop whose execution size is not bounded by a constant is the **for** loop associated with pop actions. This loop iterates over a set Z that contains the successors of some RCG node q. Since at Step i there are at most $O(i)$ RCG nodes, Z has size at most $O(n)$. Thus the order of the algorithm is at most $O(n^3)$.

8 A cubic RIGLR parser

It is in the nature of RIGLR recognisers that non-recursive instances of nonterminals are effectively substituted by the right hand sides of their grammar rules. Thus some of the hierarchical structure embodied in the grammar is flattened. Furthermore, if right recursion is not removed then the RCAcontains a loop labelled with the corresponding reduction, and the recogniser traverses this loop only once, regardless of the actual number of recursive reductions performed in the corresponding derivation. (See [21] more specific details of this.) For these reasons the construction of a parser version of the RIGLR algorithm is not straightforward. We have given an RIGLR parser which can be correctly applied to all context free grammars and which produces Tomita style SPPFs [21]. However, as implied by Johnson's observation [13], the parsing algorithm has unbounded polynomial order. In fact, to create the parser the corresponding SPPF contexts are attached to each element of U_i, substantially increasing the sizes of these sets.

In this section we give a different method for constructing derivations which results in a parser which is worst-case cubic. To efficiently identify when a nonterminal node in the SPPF should be constructed we require that RCA(Γ) has been constructed using a terminalised version of Γ in which all nonterminal instances are terminalised. We call RCA(Γ) *maximally terminalised* if Γ_S has no non-terminalised right hand side nonterminal instances.

Recall from Section 2 that SPPF nodes can have families of children, each family corresponding to a different sub-SPPF but deriving the same potion of the input string. In the case where a node w has more than one family of children then each family has a parent packed node which is a child of w. As for the Earley parser described above, the RIGLR parser will construct binarised SPPFs in which intermediate nodes are used, so that packed nodes have at most two children, a right hand symbol node and, possibly, a left hand symbol or intermediate node.

We assume that the RCAis given in the form of a table, as described in Section 7. However, we only need the left hand sides of reductions so these are stored in the form $\mathcal{R}(A, k)$.

When an input symbol is read or a reduction is performed then a corresponding SPPF node is constructed. Each descriptor $(h, q, w) \in U_i$ contains a *current context* SPPF node w. When two or more symbols, $x_1 \dots x_f x$ say, of a rule have been recognised then an intermediate SPPF node is constructed. If $(x_1 \dots x_f, j, k)$ is the label of the current context node and (x, k, i) labels the node just constructed then the intermediate node is labelled $(x_1 \dots x_f x, j, i)$ and becomes the current context node for the new descriptor.

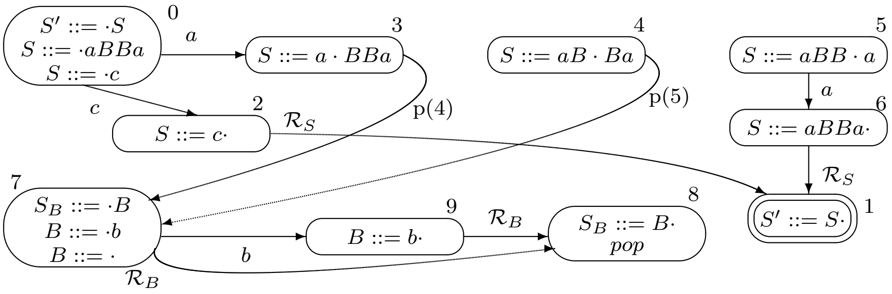
As discussed in detail in [21], we need to be careful when performing push actions in the parser. In an RIGLR recogniser, when two descriptors have the same RCG node then their call stacks are recombined, to prevent the size of the RCG from exploding. However, if there are two different SPPF context nodes associated with these descriptors then the information about which node belongs to which descriptor will be lost, resulting in the incorrect combination of the first half of one derivation with the second half of the other. (It is this effect that, as mentioned in Section 1, causes the outline parser given in [3] to fail in some cases and is analogous to the problem with Earley's original parser.) To avoid this problem, when a push action is performed the RCG edge created is labelled with the current context SPPF node, and then the current context node is set to *null*. When the RCG node is subsequently popped, the corresponding SPPF context node is retrieved.

Before giving the formal RIGLR cubic parsing algorithm we give two examples, the first illustrating the basic process and the second illustrating the subtlety of the application of pop actions.

Example 5 Consider the string aba and the grammar Γ_6

$$S \ ::= \ a\ B\ B\ a \mid c \qquad B \ ::= \ b \mid \epsilon$$

which has maximally terminalised RCA

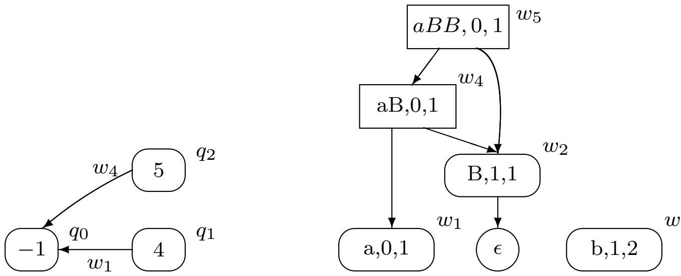


We begin by creating an RCG node q_0 labelled -1 and a descriptor $(0, q_o, null)$ that we add to U_0. The first input symbol is a and the RCAstate 0 has an a-transition to state 3. Thus we create an SPPF node w_1 labelled $(a, 0, 1)$ and add $(3, q_0, w_1)$ to U_1. All the descriptors in U_0 have now been processed so Step 0 is complete.

Next we process $(3, q_0, w_1)$. State 3 has a transition labelled $p(4)$ to state 7, thus we create an RCG node q_1 labelled 4 and an edge (q_1, w_1, q_0) from q_1 to q_0 labelled w_1, and add the descriptor $(7, q_1, null)$ to U_1. Next we process $(7, q_1, null)$, applying the action $\mathcal{R}_B$. Since the SPPF context is *null* we create an SPPF node w_2 labelled $(B, 1, 1)$ with family (ϵ) and add the descriptor $(8, q_1, w_2)$ to U_1. We also apply the b-transition from state 7, creating an SPPF node w_3 labelled $(b, 1, 2)$ and adding the descriptor $(9, q_1, w_3)$ to U_2.

From $(8, q_1, w_2)$ we apply the pop action. The node q_1 has an edge to q_0 which is labelled w_1, indicating that the 'cached' SPPF context was w_1. Thus we create a new SPPF intermediate node w_4 labelled $(aB, 0, 1)$ with family (w_1, w_2). The node w_4 is the new SPPF context and we create the descriptor $(4, q_0, w_4)$, because 4 is the label of q_1.

Processing $(4, q_0, w_4)$ we create a new RCG node q_2 labelled 5 and an edge (q_2, w_4, q_0), and add $(7, q_2, null)$ to U_1. Processing this descriptor, for the b-transition we find that there is already a node w_3 labelled $(b, 1, 2)$ and so we add $(9, q_2, w_3)$ to U_2. For $\mathcal{R}_B$ we already have the node w_2 labelled $(B, 1, 1)$ so we create the descriptor $(8, q_2, w_2)$. From the pop action for this descriptor we create a new intermediate SPPF node w_5 labelled $(aBB, 0, 1)$ with family (w_4, w_2), and add $(5, q_0, w_5)$ to U_1. This descriptor has no applicable actions so Step 1 is complete.



$$U_1 = \{(3, q_0, w_1), (7, q_1, null), (8, q_1, w_2), (4, q_0, w_4), (7, q_2, null), (8, q_2, w_2), (5, q_0, w_5)\}$$
$$U_2 = \{(9, q_1, w_3), (9, q_2, w_3)\}$$

Processing $(9, q_1, w_3)$ we apply the $\mathcal{R}_B$ action, creating an SPPF node w_6 labelled $(B, 1, 2)$ with family (w_3) and descriptor $(8, q_1, w_6)$. From $(9, q_2, w_3)$ we create $(8, q_2, w_6)$. From $(8, q_1, w_6)$ we create an intermediate node w_7, labelled $(aB, 0, 2)$ with family (w_1, w_6) and add the descriptor $(4, q_0, w_7)$ to U_2. Similarly, from $(8, q_3, w_6)$ we create an intermediate node w_8 labelled $(aBB, 0, 2)$ with family (w_4, w_6), and a descriptor $(5, q_0, w_8)$.

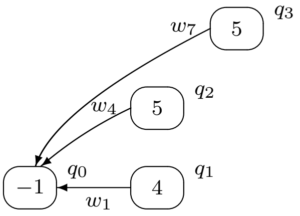
From $(4, q_0, w_7)$ we create an RCG node q_3 labelled 5, an edge (q_3, w_7, q_0) and a descriptor $(7, q_3, null)$. From $(5, q_0, w_8)$ we create an SPPF node w_{10} labelled $(a, 2, 3)$ and an intermediate node w_{11} labelled $(aBBa, 0, 3)$ with family (w_8, w_{10}), and add $(6, q_0, w_{11})$ to U_3.

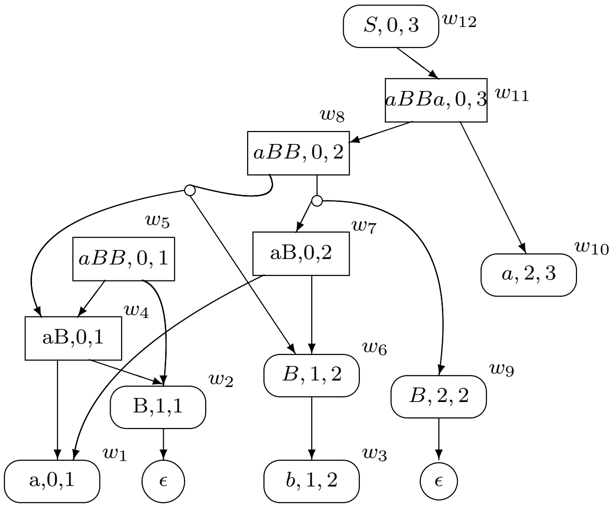
From $(7, q_3, null)$ we create an SPPF node w_9 labelled $(B, 2, 2)$ with family ϵ and a descriptor $(8, q_3, w_9)$. There is already an intermediate node w_8 labelled $(aBB, 0, 2)$ so we add the family (w_7, w_9) to w_8. The descriptor $(5, q_0, w_8)$ already exists, so Step 2 is complete.

$$U_2 = \{(9, q_1, w_3), (9, q_2, w_3), (8, q_1, w_6), (8, q_2, w_6), (4, q_0, w_7), (5, q_0, w_8), (7, q_3, null), (8, q_3, w_9)\}$$

$$U_3 = \{(6, q_0, w_{11})\}$$

Finally we process $(6, q_0, w_{11})$ and apply the $\mathcal{R}_S$ action, creating an SPPF node w_{12} labelled $(S, 0, 3)$ with family (w_{11}) and the descriptor $(1, q_0, w_{12})$. Step 3 is now complete and, since U_3 contains the descriptor $(1, q_0, w_{12})$ and 1 is the RCAaccepting state, the algorithm reports success.

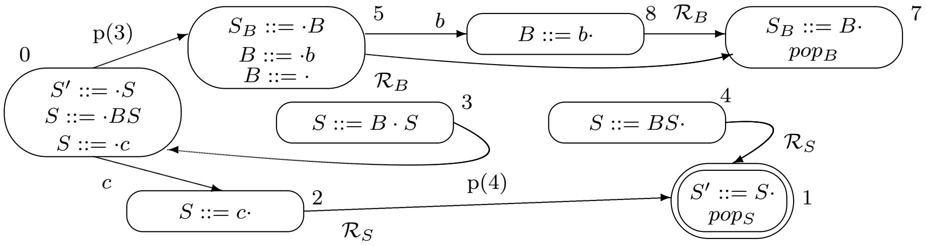




Example 6 Consider the string bc and the grammar Γ_7

$$S \ ::= \ B\ S \mid c \qquad B \ ::= \ b \mid \epsilon$$

which has maximally terminalised RCA

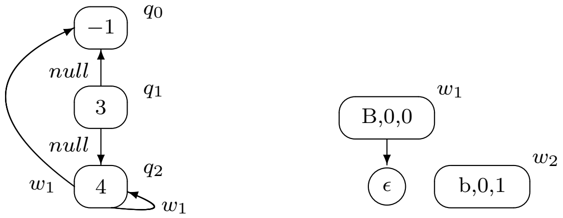


Recall that the RIGLR recogniser uses sets P_i to record RCG nodes, constructed at the current step, that have been popped. In the parser version we need the SPPF contexts associated with the pops in order to construct the new SPPF node. Thus P_i is a set of pairs (q, L) where q is an RCG node and L is a set of SPPF nodes which are the contexts in which q has been popped.

We begin, as before, by creating an RCG node q_0 labelled -1 and a descriptor $(0, q_o, null)$. We then apply the action $p(3)$, creating an RCG node q_1 labelled 3, edge $(q_1, null, q_0)$ and descriptor $(5, q_1, null)$. From the $\mathcal{R}_B$ action we create an SPPF node w_1 labelled $(B, 0, 0)$ with family (ϵ) and a descriptor $(7, q_1, w_1)$.

We then apply the pop action from $(7, q_1, w_1)$. Since the edge (q_1, q_0) is labelled *null*, we create the descriptor $(3, q_0, w_1)$. Since q_1 was created at this step we record that it has been popped with SPPF context node w_1 by adding $(q_1, \{w_1\})$ to P_0. Since state 5 also has a b-transition we create w_2 labelled $(b, 0, 1)$ and add $(8, q_1, w_2)$ to U_1.

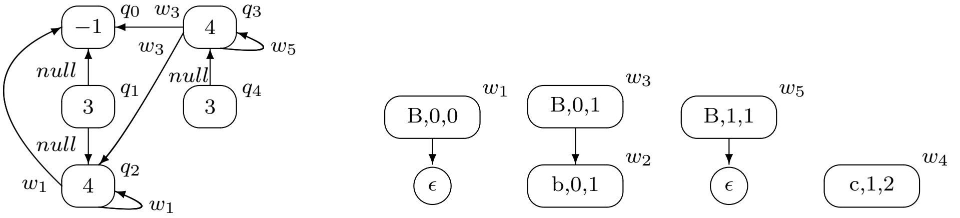
Processing $(3, q_0, w_1)$ we create q_2 labelled 4, an edge (q_2, w_1, q_0) and a descriptor $(0, q_2, \mathit{null})$. There already exists an RCG node q_1 labelled 3, so we add an edge $(q_1, \mathit{null}, q_2)$. The descriptor $(5, q_1, \mathit{null})$ already exists but we have added an edge to q_1. Since $(q_1, \{w_1\}) \in P_0$ we create the descriptor $(3, q_2, w_1)$. From $(3, q_2, w_1)$ we add an edge (q_2, w_1, q_3), and Step 0 is complete.



$$U_0 = \{(0, q_0, \mathit{null}), (5, q_1, \mathit{null}), (7, q_1, w_1), (3, q_0, w_1), (0, q_2, \mathit{null}), (3, q_2, w_1)\}$$
$$P_0 = \{(q_1, w_1)\} \qquad U_1 = \{(8, q_1, w_2)\}$$

From $(8, q_1, w_2)$ we apply $\mathcal{R}_B$, creating w_3, labelled $(B, 0, 1)$ with family (w_2), and $(7, q_1, w_3)$. We then apply the pop action and create $(3, q_0, w_3)$ and $(3, q_2, w_3)$. Applying $p(4)$ from state 3, we create q_3 labelled 4, edges (q_3, w_3, q_0) and (q_3, w_3, q_2) and descriptor $(0, q_3, \mathit{null})$. We then create q_4 labelled 3, the edge $(q_4, \mathit{null}, q_3)$ and the descriptor $(5, q_4, \mathit{null})$. Applying the c-transition we create an SPPF node w_4 labelled $(c, 1, 2)$ and add $(2, q_3, w_4)$ to U_2.

From $(5, q_4, \mathit{null})$ we create w_5 labelled $(B, 1, 1)$ with family (ϵ) and the descriptor $(7, q_4, w_5)$. The pop action results in $(3, q_3, w_5)$ and then $p(4)$ results in an edge (q_3, w_5, q_3).

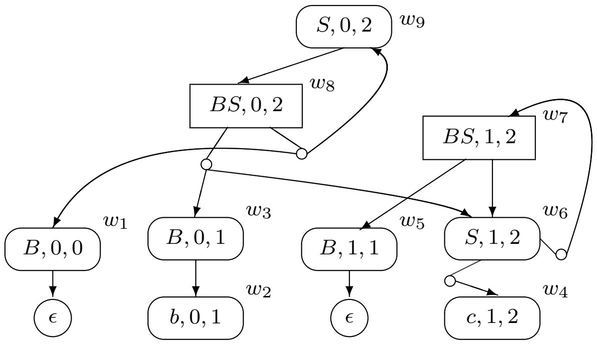


$$U_1 = \{(8, q_1, w_2), (7, q_0, w_3), (3, q_0, w_3), (3, q_1, w_1), (0, q_3, \mathit{null}), (5, q_4, \mathit{null}), (7, q_4, w_5), (3, q_3, w_5)\}$$
$$P_1 = \{(q_4, w_5)\} \qquad U_2 = \{(2, q_3, w_4)\}$$

Finally, from $(2, q_3, w_4)$ we create w_6 labelled $(S, 1, 2)$ with family (w_4) and $(1, q_3, w_6)$, and then the intermediate nodes w_7 labelled $(BS, 1, 2)$ with family (w_5, w_6) and w_8 labelled $(BS, 0, 2)$ with family (w_3, w_6). This generates descriptors $(4, q_3, w_7)$, $(4, q_0, w_8)$ and $(4, q_2, w_8)$.

From $(4, q_3, w_7)$ we add a new family (w_7) to the SPPF node w_6, but the descriptor $(1, q_3, w_6)$ is already in U_2. From $(4, q_0, w_8)$ we create an SPPF node

w_9 labelled $(S, 0, 2)$ with family (w_8) and add $(1, q_0, w_9)$ to U_2. From $(4, q_2, w_8)$ we add $(1, q_2, w_9)$ to U_2 and from this descriptor we add a new family (w_1, w_9) to w_8, but no new descriptors are created. Then Step 2 is complete and, since $(1, q_0, w_9) \in U_2$, the algorithm reports success.



An RIGLR parser

```
input: a maximally terminalised RCA written as a table $T$, and a string $a_1 \dots a_n$
define $a_{n+1} = \$$, and set $U_0, \dots, U_n$ and $P_0, \dots, P_n$ to $\emptyset$
create a base node, $q_0$, in the RCG and add $(q_0, \emptyset)$ to $P_0$
create an element, $u_0 = (0, q_0, null)$, in $U_0$
for $i = 0$ to $n$ do {
  create an SPPF node $y_i$ labelled $(a_i, i, i+1)$
  add all the elements of $U_i$ to $\mathcal{A}$
  while $\mathcal{A} \neq \emptyset$ {
    remove $u = (h, q, w)$ from $\mathcal{A}$
    if $sk \in T(h, a_{i+1})$ { $y := createT(w, i)$
                              add $(k, q, y)$ to $U_{i+1}$ }
    if $\mathcal{R}(X, k) \in T(h, a_{i+1})$ { $y := createN(w, X)$
                                    add $(k, q, y)$ to $U_{i+1}$ }
    if $pop \in T(h, a_{i+1})$ {
        let $k$ be the label of $q$
        $Z$ be the set of edge labels and successors of $q$
        if $(q, null) \in P_i$ replace it with $(q, w)$
        for each $(z, p) \in Z$ {
          $y := createI(z, w)$
          if $(k, p, y) \notin U_i$ add $(k, p, y)$ to $\mathcal{A}$ and $U_i$ } }
    for each $p(l, k) \in T(h, a_{i+1})$ {
        if there is $(t, z) \in P_i$ such that $t$ has label $l$ {
          if there is no edge from $t$ to $q$ labelled $w$ {
            add an edge from $t$ to $q$ labelled $w$
            if  $z \neq null$ {
              $y := createI(w, z)$
```

```
                **if** $(k,p,y) \notin U_i$ add $(k,p,y)$ to $\mathcal{A}$ and $U_i$ } } }
            **else** {
              create an RCG node $t$ with label $l$
              create an edge from $t$ to $q$ labelled $w$
              add $(k,t,null)$ to $\mathcal{A}$ and $U_i$ and add $(t,null)$ to $P_i$ } } }
}
if $U_n$ contains $(l,q_0,w)$ where $l$ is an accept state of the RCA {
      remove each intermediate node with no siblings by copying its out-edges to its parents
      report success }
else { report failure }

$createT(w,i)$ {
  **if** $(w = null)$ { let $y := y_i$ }
  **else** {
    suppose that $w$ has label $(\mu,j,i)$
    **if** there does not exist an SPPF node $y$ labelled $(\mu a_{i+1},j,i+1)$ create one
    **if** $y$ does not have the family $(w,y_i)$  add the family $(w,y_i)$ to $y$ }
  return $y$ }

$createN(w,X)$ {
  **if** $(w = null)$ { let $k := i$ and $w := \epsilon$ }
  **else** { suppose that $w$ has label $(\mu,k,i)$ }
  **if** there does not exist an SPPF node $y$ labelled $(X,k,i)$ create one
  **if** $y$ does not have a family $(w)$ add one }
  return $y$ }

$createI(w,z)$ {
  suppose that $z$ has label $(X,k,i)$
  **if** $w = null$ { let $y := z$ }
  **else** { suppose that $w$ has label $(\nu,j,k)$
      **if** there does not exist an SPPF node $y$ labelled $(\nu X,j,i)$ create one
      **if** $y$ does not have a family $(w,z)$ add one }
  return $y$ }
```

9 Summary and conclusions

In this paper we have given a correct, worst-case cubic, parser based on Earley's recognition algorithm, a proof that the RIGLR recognisers are worst-case cubic, and a worst-case cubic RIGLR parser. Both parsers construct a binarised SPPF that represents all possible derivations of the given input string. The approach is based on the approach taken in BRNGLR, a cubic version of Tomita's algorithm, and the SPPFs constructed are equivalent to those constructed by BRNGLR.

This paper has two goals, to present a correct general Earley parser and to highlight the fact that constructing a parsing algorithm can be more a difficult and subtle process than constructing the corresponding recogniser. The parser extensions of the Earley and RIGLR algorithms discussed in [7] and [3], respectively, construct spurious derivations. Tomita's GLR recogniser, constructed explicity as a first step to a parser and thus effectively embedding an SPPF, was not general and the Farshi [18] and RNGLR [22] corrections both have unbounded polynomial order.

We can modify Earley's proposed parser so that it does produce correct derivations by labelling the pointers he introduces, see [20] for more details. However, this does not result in a cubic parser. To achieve this we have effectively introduced item pointers to the underlying recogniser, reflected in the SPPF nodes associated with the Earley items.

For an efficient RIGLR parser, we have had to require that RCA(Γ) be constructed using a maximally terminalised version of Γ. Thus in order to construct a cubic parser we have had to modify the underlying recogniser, which would normally have fewer push actions. This mirrors the situation for GLR parsers in which the cubic BRNGLR recogniser creates more GSS states than the corresponding RNGLR recogniser. However, in both cases the cubic versions require less searching effort and this is how the asymptotic improvement is obtained.

References

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

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

[3] John Aycock, R. Nigel Horspool, Jan Janousek, and Borivo Melichar. Even faster generalised LR parsing. *Acta Informatica*, 37(8):633–651, 2001.

[4] Claus Brabrand. *Grambiguity.* `http://www.brics.dk/ brabrand/grambiguity/`, 2006.

[5] Frank L DeRemer and Thomas J. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Progam. Lang. Syst.*, 4(4):615–649, October 1982.

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

[7] J Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition.* Addison-Wesley, 2005.

[10] Susan L. Graham and Michael A. Harrison. Parsing of general context-free languages. *Advances in Computing*, 14:77–185, 1976.

[11] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide.* Monographs in Computer Science. Springer, Berlin, 2008.

[12] John E Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Series in Computer Science. Addison-Wesley, 1979.

[13] Mark Johnson. The computational complexity of GLR parsing. In Masaru Tomita, editor, *Generalized LR parsing*, pages 35–42. Kluwer Academic Publishers, The Netherlands, 1991.

[14] Adrian Johnstone and Elizabeth Scott. Proofs and pedagogy; science and systems: the Grammar Tool Box. 69:76–85, 2007.

[15] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Generalised parsing: some costs. In Evelyn Duesterwald, editor, *Compiler Construction, 13th Intnl. Conf, CC'04*, volume 2985 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, Berlin, 2004.

[16] Donald E Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[17] Scott McPeak and George Necula. Elkhound: a fast, practical GLR parser generator. In Evelyn Duesterwald, editor, *Compiler Construction, 13th Intnl. Conf, CC'04*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2004.

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

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

[20] Elizabeth Scott. SPPF-style parsing from Earley recognisers. *Electronic Notes in Theoretical Computer Science*, pages 53–67, 2007.

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

[22] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, July 2006.

[23] Elizabeth Scott, Adrian Johnstone, and Giorgios Economopoulos. A cubic Tomita style GLR parsing algorithm. *Acta Informatica*, 44(6):427–461, 2007.

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

[25] D H Younger. Recognition of context-free languages in time n^3. *Inform. Control*, 10(2):189–208, February 1967.