Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 203 (2008) 53–67
www.elsevier.com/locate/entcs

SPPF-Style Parsing From Earley Recognisers

Elizabeth Scott[1]

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

Abstract

In its recogniser form, Earley's algorithm for testing whether a string can be derived from a grammar is 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. We have given a modified worst-case cubic version, the BRNGLR 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 two versions of an Earley parsing algorithm that, in worst-case cubic time, return an SPPF representation of all derivations of a given string from a given CFG.

Keywords: Earley parsing, cubic generalised parsing, context free languages

Since Knuth's seminal 1960's work on LR parsing [14] was extended to LALR parsers by DeRemer [5,4], 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.

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 [7] 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 [8] the compiler version of the grammar is written in EBNF and is (unnecessarily) ambiguous, il-

[1] Email:e.scott@rhul.ac.uk

doi:10.1016/j.entcs.2008.03.044

lustrating the difficulty of making correct transformations. Given this difficulty in constructing natural LR(1) grammars that support desired semantics, the general parsing techniques, such as the CYK [20], Earley [6] and GLR [19] 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 and Earley algorithms are recognisers while 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 is of unbounded polynomial order in worst case. In this paper we describe the expansion of Earley 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 cubic on grammars in Chomsky normal form and Earley's algorithm is worst case cubic on general context free grammars and worst case order n^2 on non-ambiguous grammars. 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. A cubic recogniser which was modified to simply return all derivations could become an unbounded parser.

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 statement in the ANSI-standard grammar for C 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 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 [11], 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, [3], but this remains a topic requiring further research.

Another possibility is to avoid the issue by just returning one derivation. In [9] 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 [19]. 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 [15] of Tomita's GLR algorithm, was given by Rekers [16]. These algorithms are all worst-case unbounded polynomial order and, in fact, Johnson [12] 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.

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 [19] to return spurious derivations in certain cases.

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.

In [2] there is given an outline of an algorithm to turn the recogniser reported there and in [1] into a parser, but again, as written, this algorithm will generate spurious derivations as well as the correct ones. The recogniser described in [1] is not applicable to grammars with hidden left recursion but the closely related RIGLR algorithm [18] is fully general, and 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.

As we have mentioned, Tomita's GLR algorithm was designed with parse tree construction in mind. We have given a GLR algorithm, BRNGLR [17], 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 recogniser and construct two versions of a complete Earley parser, both of which are worst case cubic order. Thus 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.

2 Background theory

In this section we give a brief description of Earley's algorithm, for simplicity without lookahead, and show how Earley's own extension of this to a parser can fail.

We then show how to apply the techniques developed in [17] to correctly generate a representation of all possible derivations of a given input string from Earley's recogniser in worst case cubic time and space.

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 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 are labelled with a triple (x, j, i) where $a_{j+1} \ldots a_i$ is a substring matched by x. To obtain a cubic algorithm we use *binarised* SPPFs which contain intermediate *additional* 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 $a_1 \ldots a_n$, a set of *items*. Each item represents a position in the grammar that a top down parser could be in after matching $a_1 \ldots a_i$. 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 $(S ::= \alpha\cdot, 0) \in \mathbf{E}_n$.

As an example consider the grammar

$$S ::= ST \mid a \qquad B ::= \epsilon \qquad T ::= aB \mid a$$

and input string aa. The Earley sets are

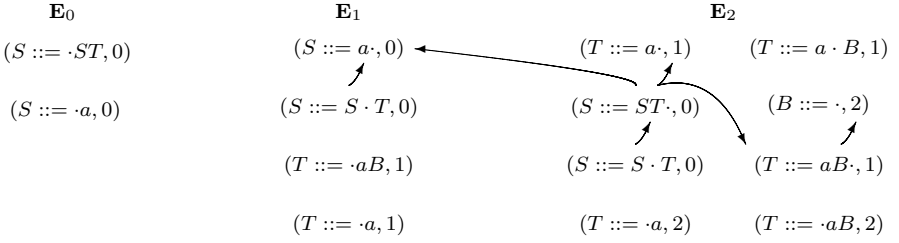
$$\begin{aligned}
\mathbf{E}_0 &= \{(S ::= \cdot ST, 0), (S ::= \cdot a, 0)\} \\
\mathbf{E}_1 &= \{(S ::= a\cdot, 0), (S ::= S \cdot T, 0), (T ::= \cdot aB, 1), (T ::= \cdot a, 1)\} \\
\mathbf{E}_2 &= \{(T ::= a \cdot B, 1), (T ::= a\cdot, 1), (B ::= \cdot, 2), (S ::= ST\cdot, 0), \\
&\qquad (T ::= aB\cdot,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

3 Problems with Earley parser construction

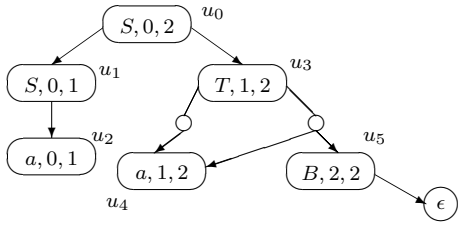
Earley's original paper gives a brief description of how to construct a representation of all possible derivation trees from the recognition algorithm, and claims that this requires at most cubic time and space. The proposal is to maintain pointers from the non-terminal instances on the right hand sides 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 we add pointers 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 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 the level 2 Earley set, 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 under a new packed node and create a further packed node with child u_4.

The procedure proposed by Earley works correctly for the above example, but adding multiple pointers to a given instance of a non-terminal can create errors. As remarked in [19] p74, if we consider the grammar

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

and the input string *bbb* we find that the above procedure generates the correct derivations of *bbb* but also spurious derivations of the strings *bbbb* and *bb*. The problem is that the derivation of *bb* from the left-most S in one derivation of *bbb* becomes intertwined with the derivation of *bb* from the rightmost S in the other derivation, resulting in the creation of *bbbb*.

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 of the two items. 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

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

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 [10] has described a parser 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 be of at least unbounded polynomial order in worst case.

4 A cubic parser which walks the Earley sets

We can turn Earley's algorithm into a correct parser by adding pointers between items rather than instances of non-terminals, and labelling the pointers in a way which allows a binarised SPPF to be constructed by walking the resulting structure. (In the next section we shall give a version of the algorithm that constructs a binarised SPPF as the Earley sets are constructed.)

Set $\mathbf{E}_0$ to be the items $(S ::= \cdot\alpha, 0)$. For $i > 0$ initialise $\mathbf{E}_i$ by adding the item $p = (A ::= \alpha a_i \cdot \beta, j)$ for each $q = (A ::= \alpha \cdot a_i\beta, j) \in \mathbf{E}_{i-1}$ and, if $\alpha \neq \epsilon$, creating a predecessor pointer labelled $i-1$ from q to p. Before initialising $\mathbf{E}_{i+1}$ complete $\mathbf{E}_i$ as follows. For each item $(B ::= \gamma \cdot D\delta, k) \in \mathbf{E}_i$ and each rule $D ::= \rho$, $(D ::= \cdot\rho, i)$ is added to $\mathbf{E}_i$. For each item $t = (B ::= \tau\cdot, k) \in \mathbf{E}_i$ and each corresponding item $q = (D ::= \tau \cdot B\mu, h) \in \mathbf{E}_k$, if there is no item $p = (D ::= \tau B \cdot \mu, h) \in \mathbf{E}_i$ create one. Add a reduction pointer labelled k from p to t and, if $\tau \neq \epsilon$, a predecessor pointer labelled k from p to q.

We could walk the above structure in a fashion that is essentially the same as described in Example 1 above. However, in order to construct a binarised SPPF we also have to introduce additional nodes for grammar rules of length greater than two. Thus the final algorithm is slightly more complicated.

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 one or two nodes. For a symbol node the family will correspond to a grammar rule $B ::= \gamma y$ or $B ::= \epsilon$. If $\gamma \neq \epsilon$ then the children will be labelled $(B ::= \gamma \cdot y, j, l)$ and (y, l, i), for some l. Otherwise there will be a single child in the family, labelled (y, j, i) or ϵ. For an additional node the family will have a child labelled (x, l, i). If $\gamma \neq \epsilon$ then the family will have a second child labelled $(B ::= \gamma \cdot x\delta, j, l)$.

We now define a function which takes an SPPF node u and an item p from an Earley set $\mathbf{E}_i$, possibly decorated with pointers, and builds the corresponding part of the SPPF. A decorated item consists of a LR(0)-item, $A ::= \alpha \cdot \beta$, a left hand index j, a right hand index, i, and a set of associated labelled pointers. We assume that these attributes and the complete Earley set structure are passed into Buildtree with u and p.

Buildtree(u,p) {
suppose that $p \in \mathbf{E}_i$ and that p is of the form $(A ::= \alpha \cdot \beta, j)$
mark p as processed

if $p = (A ::= \cdot, j)$ {
if there is no SPPF node v labelled (A, i, i)
create one with child node ϵ
if u does not have a family (v) then add the family (v) to u }

if $p = (A ::= a \cdot \beta, j)$ (where a is a terminal) {
if there is no SPPF node v labelled $(a, i-1, i)$ create one
if u does not have a family (v) then add the family (v) to u }

if $p = (A ::= C \cdot \beta, j)$ (where C is a non-terminal) {
if there is no SPPF node v labelled (C, j, i) create one
if u does not have a family (v) then add the family (v) to u
for each reduction pointer from p labelled j {
suppose that the pointer points to q
if q is not marked as processed Buildtree(v,q) } }

if $p = (A ::= \alpha' a \cdot \beta, j)$ (where a is a terminal, $\alpha' \neq \epsilon$) {
if there is no SPPF node v labelled $(a, i-1, i)$ create one
if there is no SPPF node w labelled $(A ::= \alpha' \cdot a\beta, j, i-1)$ create one
for each target p' of a predecessor pointer labelled $i-1$ from p {
if p' is not marked as processed Buildtree(w,p') }
if u does not have a family (w, v) add the family (w, v) to u }

if $p = (A ::= \alpha' C \cdot \beta, j)$ (where C is a non-terminal, $\alpha' \neq \epsilon$) {
for each reduction pointer from p {
suppose that the pointer is labelled l and points to q
if there is no SPPF node v labelled (C, l, i) create one
if q is not marked as processed Buildtree(v,q)
if there is no SPPF node w labelled $(A ::= \alpha' x \cdot C\beta, j, l)$ create one
for each target p' of a predecessor pointer labelled l from p {
if p' is not marked as processed Buildtree(w,p') }
if u does not have a family (w, v) add the family (w, v) to u } }
}

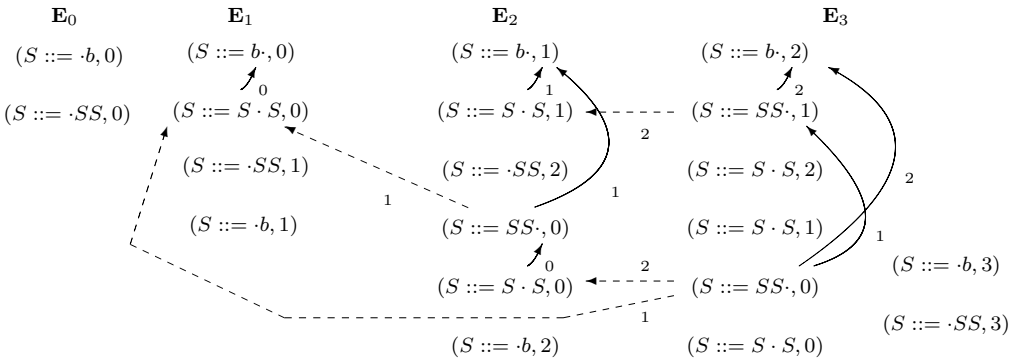
We build the full SPPF from the root down using the following procedure.

PARSER { create an SPPF node u_0 labelled $(S, 0, n)$
for each decorated item $p = (S ::= \alpha\cdot, 0) \in \mathbf{E}_n$ Buildtree(u_0, p) }

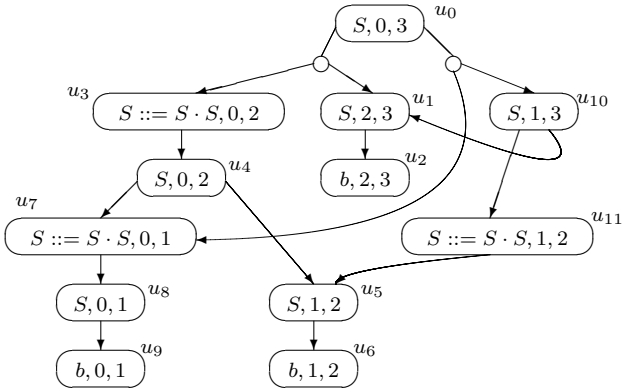
We illustrate this approach using two examples: the first is the example, discussed above, that results in an error when Earley's parsing approach is used; and the second is a grammar with hidden left recursion and a cycle, resulting in infinitely many derivations.

Example 2 Grammar : $S ::= S\ S \mid b$ Input : *bbb*

The Earley set structure is essentially



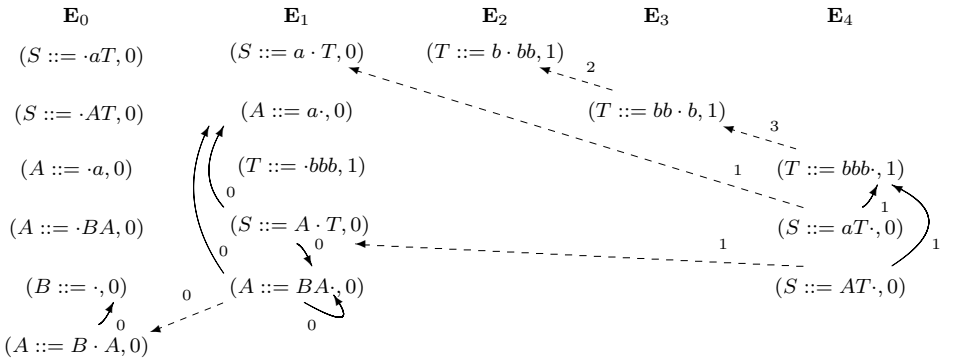
(for ease of reading pointers from nodes not reachable from the node in $\mathbf{E}_3$ labelled $(S ::= SS\cdot, 0)$ have been left off the diagram). The corresponding (correct) binarised SPPF, with the nodes labelled in construction order, is



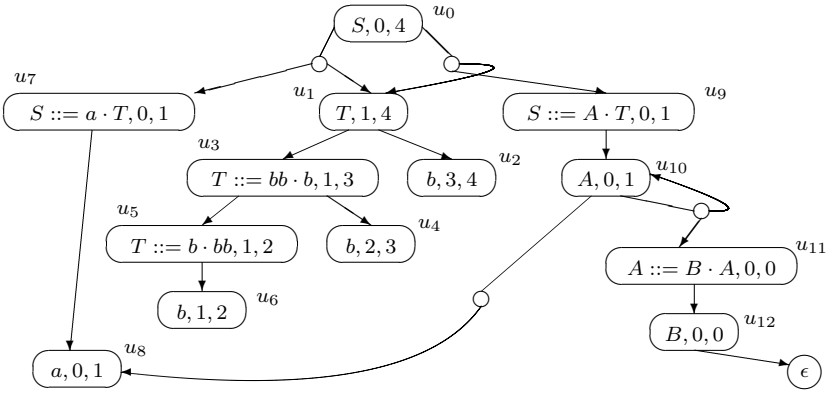
Example 3

Grammar : $S ::= A\ T \mid a\ T \quad A ::= a \mid B\ A \quad B ::= \epsilon \quad T ::= b\ b\ b$
Input : *abbb*

The Earley set structure is



and the corresponding binarised SPPF is



5 An integrated parsing algorithm

The Buildtree function described in the previous section is not as efficient as it could be, it has been designed to reflect the principles underlying the approach. We now give a different version of an Earley parser that constructs a binarised SPPF as the Earley sets are constructed, and does not require the items to be decorated with 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. It is also slightly smaller than the corresponding SPPF from the previous section as a node with a label of the form $(A ::= x \cdot \beta, j, i)$ is merged with its child.

The algorithm itself is in a form that is similar to the form 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 each Earley item the SPPF node that corresponds to it. Thus Earley items are triples (s, j, w) where s is a non-terminal or an LR(0) item, j is an integer and w is an SPPF node with a label of the form (s, j, l). The subtree below such a node w will

correspond to the derivation of the substring $a_{j+1} \dots a_l$, from B if s is B and from α if s is $B ::= \alpha \cdot \beta$. Earley items of the form $(A ::= \alpha \cdot \beta, j)$ where $|\alpha| \leq 1$ 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 $A \Rightarrow \alpha \stackrel{*}{\Rightarrow} \epsilon$. 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 an item of this form 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, w) 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, ϵ.

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

EARLEY_PARSER {

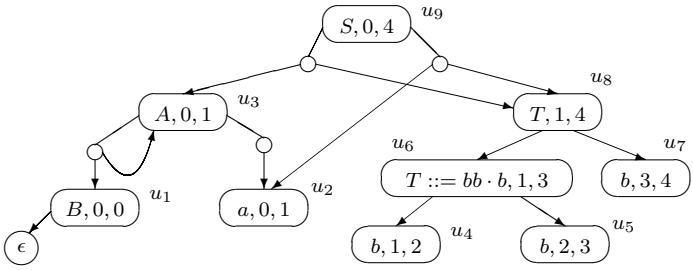
$\mathbf{E}_0, \dots, \mathbf{E}_n$, $\mathcal{R}$, $\mathcal{Q}'$, $V = \emptyset$
for all $(S ::= \alpha) \in \mathcal{P}$ {
  if $\alpha \in \Sigma_N$ add $(S ::= \cdot\alpha, 0, null)$ to $\mathbf{E}_0$
  if $\alpha = a_1\alpha'$ add $(S ::= \cdot\alpha, 0, null)$ to $\mathcal{Q}'$ }

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, Λ 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 β = a_{i+1}β' { add (B ::= αC · β, h, y) to Q } } }
      if Λ = (D ::= α·, h, w) {
        if w = null {
            if there is no node v ∈ V labelled (D, i, i) create one
            set w = v
            if w does not have family (ε) add one }
        if h = i { add (D, w) to H }
        for all (A ::= τ · Dδ, k, z) in E_h {
            let y = MAKE_NODE(A ::= τD · δ, k, i, z, w, V)
            if δ ∈ Σ_N and (A ::= τD · δ, k, y) ∉ E_i {
                        add (A ::= τD · δ, k, y) to E_i and R }
            if δ = a_{i+1}δ' { add (A ::= τD · δ, k, y) to Q } } }
  }
  V = ∅
  create an SPPF node v labelled (a_{i+1}, i, i + 1)
  while Q ≠ ∅ {
    remove an element, Λ = (B ::= α · a_{i+1}β, h, w) say, from Q
    let y = MAKE_NODE(B ::= αa_{i+1} · β, h, i + 1, w, v, V)
    if β ∈ Σ_N { add (B ::= αa_{i+1} · β, h, y) to E_{i+1} }
    if β = a_{i+2}β' { add (B ::= αa_{i+1} · β, h, y) to Q' }
  }
}
if (S ::= τ·, 0, w) ∈ E_n return w
else return FAILURE
}

MAKE_NODE(B ::= αx · β, j, i, w, v, V) {
  if β = ε { let s = B } else { let s = (B ::= αx · β) }
  if α = ε and β ≠ ε { let y = v }
  else {
    if there is no node y ∈ V labelled (s, j, i) create one and add it to V
    if w = null and y does not have a family of children (v) add one
    if w ≠ null and y does not have a family of children (w, v) add one }
  return y
}
```

Using this algorithm on Example 3 from Section 4 results in the following SPPF.



6 The order of the parsers

(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 [17]. 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 parsing approach based on the Buildtree procedure described in Section 4, the Earley sets are constructed as for Earley's original algorithm. There are at most $O(n^2)$ items and each item has at most $O(n)$ predecessor pointers, one to each of the collections $\mathbf{E}_j$, $0 \leq j \leq i$. Because an item is marked as processed as soon as Buildtree is called on it, the parsing process makes at most $O(n^2)$ calls to Buildtree. Assuming that the SPPF is represented in a way that allows n-independent look-up time for a particular node and family of children, the only n-dependent behaviour of Buildtree occurs during the iteration over the predecessor pointers from the input item, and there are at most $O(n)$ such pointers. It is possible to represent the SPPF in the required fashion, one such representation being described in [17]. Thus our Earley parsers can be implemented so that they have worst-case cubic order.

Finally we consider the integrated Earley parser given in Section 5. 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 $O(n^3)$.

7 Summary and further work

In this paper we have given two versions of a parser based on Earley's recognition algorithm, both of which are of worst case cubic order.

Both algorithms 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. Some experimental results comparing the recogniser versions of BRNGLR and Earley's algorithm are reported in [13]. Now further experimental work is required to compare the performance of the integrated Earley parser described in this paper with the parser version of BRNGLR.

References

[1] 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.

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

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

[4] 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.

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

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

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

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

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

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

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

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

[13] 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.

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

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

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

[17] E.A. Scott, A.I.C. Johnstone, and G.R. Economopoulos. BRN-table based GLR parsers. Technical Report TR-03-06, Computer Science Department, Royal Holloway, University of London, London, 2003.

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

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

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