Elizabeth Scott Explained

Parsing from Earley Recognisers

Zoe Wheeler
University of Texas at Austin
zoe.donnellon.wheeler@gmail.com

Walter Xia
University of Texas at Austin
swilery@utexas.edu

Abstract

Earley's Algorithm is able to recognize general context-free grammars in $O(n^3)$, where n is the size of the string to be recognized. However, there are times in which we want more than just a yes or no answer. There are times in which we want an actual parse tree, and for ambiguous grammars, there are times in which we want all possible parse trees. Fortunately, there is a paper by Dr. Elizabeth Scott, [2], that presents a technique to produce a data structure known as a Shared Packed Parse Forest (SPPF), able to represent even an infinite number of parse trees. Unfortunately this paper is poorly written, making it very difficult to understand. Our paper is a re-explanation of Scott's techniques. It is agreed by many that Earley's Algorithm is also difficult to understand. Fortunately, there exists a data structure due to Dr. Gianfranco Bilardi and Dr. Keshav Pingali, [1], known as Grammar Flow Graphs (GFGs) that significantly ease the understanding of the algorithm by reformulating parsing problems as path problems in a graph. Our technique will use GFGs.

Categories and Subject Descriptors F.7.2 [*Semantics and Reasoning*]: Program Reasoning–Parsing

General Terms Context-Free Languages, Cubic Generalized Parsing, Earley Parsing

Keywords Earley Sets, Grammar Flow Graphs, Non-Deterministic Finite Automaton, Shared Packed Parse Forest

1. Introduction

It is important here for us to distinguish between recognisers and parsers for a grammar. Recognizers determine whether or not a string is part of a language defined by a grammar whereas parsers construct parse trees that reveal *how* a string satisfies the syntax dictated by a grammar. For about the past five decades, there already exist general recognizers like Cocke-Younger-Kasami (CYK) and Earley's Algorithms that run cubic relative to the size of the string to be recognized. Alternatively, Generalized LR (GLR) is an algorithm that produces parsers but has the very undesirable property that it is unbounded. Dr. Elizabeth Scott extended the Earley Recogniser into a parser that is able run in cubic space and time, [2]. The challenge was to successfully apply the parser to ambiguous grammars that produces multiple, perhaps infinite, parse trees for a string in the grammar. Note that simply disallowing ambiguous grammars is not a solution since there exists grammars that are intrinsically ambiguous. The solution she used was a representation known as a Shared Packed Parse Forest (SPPF), which is in essence a Directed Acyclic Graph (DAG).

Earley's Algorithm is a highly complex algorithm. To dramatically simplify its understanding, we view it from the perspective of Grammar Flow Graphs (GFGs) that restructure parsing as finding certain paths within the graph, [1]. For those of you familiar with automata theory, GFGs play the same role for context-free grammars as finite-state automota play for regular grammars. The rest of the paper is organized as follows:

- Section 2 will introduce GFGs
- Section 3 will introduce Earley's Algorithm using GFGs
- Section 4 will introduce SPPFs
- Section 5 will introduce Dr. Scott's Algorithm for producing SPPFs
- Section 6 will discuss our implementation
- Section 7 will discuss our results
- Section 8 will conclude

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm...$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Grammar Flow Graphs

Let us begin with the standard definition of a context-free grammar.

Definition: A *context-free grammar*, *CFG*, is a tuple (N, T, P, S), where, [1]:

- ▷ N is a finite set of elements called *nonterminals*,
- ▷ T is a finite set of elements called *terminals*,
- ▷ $P \subseteq N \times (N \cup T)^*$ is the set of *productions* that map nonterminals to a sequence of nonterminals or terminals, and
- ▷ $S \in N$ is the unique *start symbol* that appears once on the left-hand side of a single production.

An example of a grammar is the following, where | signifies or:

$$S \longrightarrow N\, t \mid t\, N$$
$$N \longrightarrow t\, t$$

Now we are in a position to introduce the GFG.

Definition: Let $CFG = (N, T, P, S)$ be a context-free grammar and let ϵ denote the empty string. The *grammar flow graph (GFG)* of CFG, $GFG(CFG) = (V(CGF), G(CFG))$, is the smalled directed graph that has the following properties, [1]:

- ▷ For each nonterminal $M \in N$, there exist $(\bullet M)$,$(M\bullet) \in V(CFG)$ called *start nodes* and *end nodes* respectively,

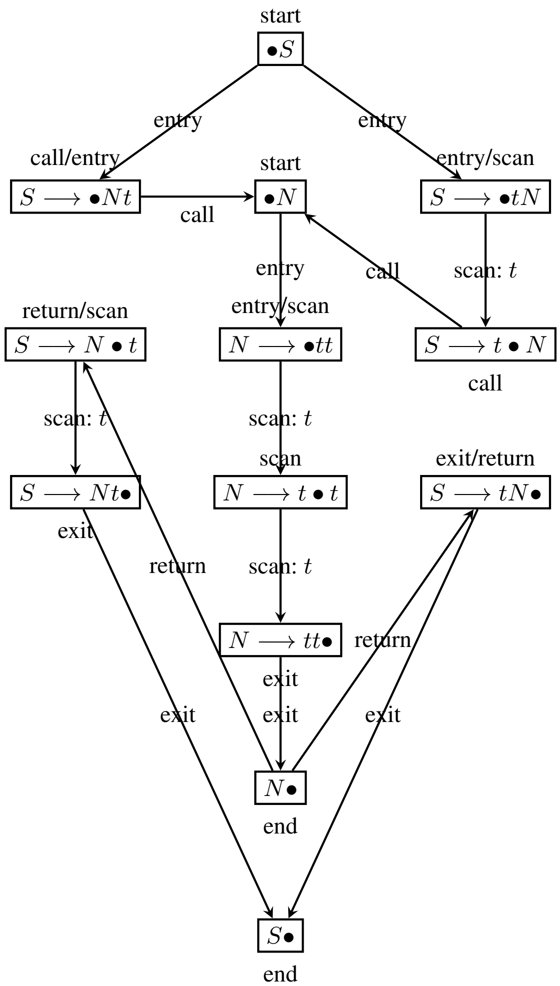

Figure 1. Example of a GFG for the preceding grammar.

- $\triangleright$ For each production $(M \longrightarrow \epsilon) \in P$, there exists $(M \longrightarrow \bullet) \in V(CFG)$ and $(\bullet M, M \longrightarrow \bullet), (M \longrightarrow \bullet, M\bullet) \in E(CFG)$,
- $\triangleright$ For each production $(M \longrightarrow q_1 q_2 \dots q_r)$ where $q_i \neq \epsilon$:
  - $\diamond$ $(M \longrightarrow \bullet q_1 q_2 \dots q_r), (M \longrightarrow q_1 \bullet q_2 \dots q_r), \dots, (M \longrightarrow q_1 q_2 ... q_r \bullet) \in V(CFG)$, where the first node is called an *entry node* and the last node is called an *exit node*,
  - $\diamond$ $(\bullet M, M \longrightarrow \bullet q_1 q_2 \dots q_r), (M \longrightarrow q_1 q_2 \dots q_r \bullet, M\bullet) \in E(CFG)$ called *entry edges* and *exit edges* respectively,
  - $\diamond$ For each $t \in T$, $(M \longrightarrow \dots \bullet t \dots, M \longrightarrow \dots t \bullet \dots) \in E(CFG)$ called *scan edges* labeled t, where $(M \longrightarrow \dots \bullet t \dots)$ is called a *scan node*, and
  - $\diamond$ For each $K \in N$, $(M \longrightarrow \dots \bullet K \dots, \bullet K)$, $(K\bullet, M \longrightarrow \dots K \bullet \dots)$ called *call edges* and *return edges* respectively, where $(M \longrightarrow \dots \bullet K \dots)$ is called a *call node* that is matched with the *return node* $(M \longrightarrow \dots K \bullet \dots)$, and
- $\triangleright$ Edges not scan edges are labeled ϵ.

Figure 1 depicts the GFG associated with the preceding grammar. The following definition comes naturally.

Definition: A path in a GFG *generates* the word w by concatenating the labels along its sequence of edges.

Those familiar with automata theory may recognize that a GFG resembles a non-deterministic finite-state automaton (NFA) which starts at $\bullet S$ and accepts at $S\bullet$. The idea is that each path from $\bullet S$ to $S\bullet$ generates a word recognized by the automaton. However, in general, this is not the case. To see this, consider the path $P = (\bullet S, S \longrightarrow \bullet tN, S \longrightarrow t \bullet N, \bullet N, N \longrightarrow \bullet tt, N \longrightarrow t \bullet t, N \longrightarrow tt\bullet, N\bullet, S \longrightarrow N \bullet t, S \longrightarrow Nt\bullet, S\bullet)$ in Figure 1. P generates the word "tttt" which is not part of the original grammar. To maintain correctness, we must restrict the valid paths the automaton can take. In the case of P, the automaton must realize that after traversing the edge $(S \longrightarrow t \bullet N, \bullet N)$ it must traverse $(N\bullet, S \longrightarrow tN\bullet)$ instead of $(N\bullet, S \longrightarrow N \bullet t)$. In general, the automaton can choose an arbitrary outgoing edge at a start node but at an end node, it must choose the return edge corresponding to the call edge it took. This behavior can be represented by a stack, by which when the automaton encounters a call node, it pushes the corresponding return node on the stack. Subsequently at an end node, the automaton pops the stack. In the case of P, $(S \longrightarrow tN\bullet)$ gets pushed on the stack at $(S \longrightarrow t \bullet N)$ and it gets popped at $N\bullet$. Dr. Bilardi and Pingali called this automaton a non-deterministic GFG automaton (NGA). We have the following definition.

Definition: The valid paths a NGA could follow from $\bullet S$ to $S\bullet$ are called *complete balanced paths (CBPs)*.

Theorem 1: Let $CFG = (N, T, P, S)$ and let $w \in T^*$. w is part of the language produced by CFG iff a CBP of GFG(CFG) generates w.

Proof: Please see [1].

3. Earley's Algorithm

Even though Earley's Algorithm is difficult to understand in the standard context, from the prospective of GFGs, it is just an algorithm that simulates the NGA. For an input string w, the algorithm generates a sequence of Earley sets, $\Sigma_0, \Sigma_1, \dots, \Sigma_{|w|}$, in which each set is a set of nodes from the GFG. Each set Σ_i is the ϵ-closure of Σ_{i-1}, that is each node in Σ_i is reachable from a node in Σ_{i-1} by traversing edges labeled ϵ in the GFG after traversing a scan edge labeled with the character at position i in string w. As its definition, Σ_0 contains $(\bullet S)$ and no characters from the string w. Intuitively, you can imagine that the characters in w start their numbering at position 1.

Recall that at an end node, the NGA should take the return edge corresponding to the call edge it took. This can be handled by associating a tag with each node in the Earley sets. At a high level, these tags differentiate the times at which the start nodes are reached and they propogate this information to the corresponding end nodes. At the end nodes, the tags are consulted to find the appropriate return edge. Thus when a call edge is traversed from a call node to a start node, the start node gets tagged with the number of the Earley Set to which the call and start nodes are added. At an end node, the tag identifies the Earley Set in which the call node resides after which the corresponding return node can be easily identified and tagged with the tag of its call node. All other edges simply copy these tags. We thus have the following theorem.

Thereom 2: Let $CFG = (N, T, P, S)$ and let w be an input string. $(S\bullet, 0) \in \Sigma_{|w|}$ iff w is part of the language produced by CFG.

Proof: Please see [1].

Figure 2 displays the Earley Sets on the input string "ttt" giving by the following grammar whose GFG is provided in Figure 1.

$$S \longrightarrow N\,t \mid t\,N$$
$$N \longrightarrow t\,t$$

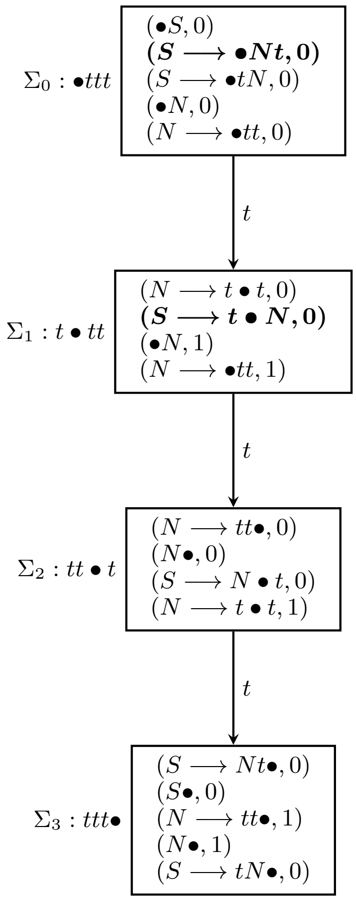

Figure 2. Example of Earley Sets for the preceding grammar on string "ttt". Call nodes are in bold.

4. Shared Packed Parse Forest (SPPF

Even though a Shared Packed Parse Forest was the purpose of her algorithm, Dr. Elizabeth Scott only gave it a brief description. Here, we will try to provide a more intuitive understanding of this data structure. First, consider the following ambiguous grammar:

$$S \longrightarrow S\,S \mid u$$

Figure 3 displays the two parse trees of the string "uuu" that is part of the above grammar. The meaning of the numbers at each node are as follows. Giving the string "uuu", number the string as shown:

$$0 \quad u \quad 1 \quad u \quad 2 \quad u \quad 3$$

So $u(0,1)$ indicates that the u of interest is between the numbers 0 and 1, $u(1,2)$ indicates that the u of interest is between the numbers 1 and 2, and so on. The numbers of the interior nodes are just natural extensions of this pattern.

Let us ignore these numbers for the moment and just consider the most natural way that allows us to share the common subtrees that appear in both parse trees in Figure 3. Notice that both subtrees consist of a root node S whose children consist of two other nodes, lets call them S_l and S_r for the moment. The difference between the two parse trees only begins to appear at level 3 in which we have two alternatives:

- Either S_l contains two children, S_m, S_s and S_r goes directly to a leaf u
- or S_l goes directly to a leaf u and S_r contains two children, S_m, S_s.

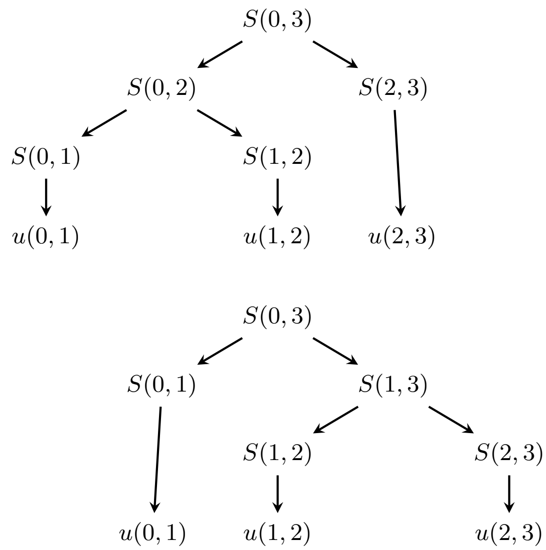

Figure 3. The two parse trees of "uuu" for the preceding grammar

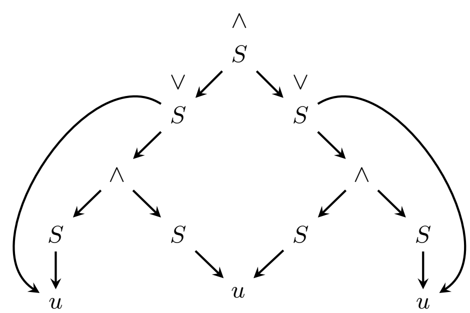

Figure 4. The and-or tree representation of the two parse trees in Figure 3.

We can capture this intuitive notion by the *and-or* tree in Figure 4, where the subscripts are ignored since they are immaterial. An *and* node represents the action that you consider both children of the node while an *or* node represents the action that you consider one child of the node. However, observe that this tree represents strings other than "uuu". To see this, consider starting at the root node and following its edges to both of its children. Here, you are at two *or* nodes both labeled S. Since we are at *or* nodes, we choose a single child for each node. Suppose we choose the leaf children, resulting in the string "uu". To correct this error, we reintroduce the numbers associated with each node and require that only nodes with identical names and numbers can share structure from the tree. Figure 5 shows our correction.

The data structure we have constructed is what Dr. Elizabeth Scott calls a Shared Packed Parse Forest (SPPF). For sake of completeness, we present her definition. Note that in her definition, her *packed* nodes are our nameless *and* nodes.

Definition: A *Shared Packed Parse Forest (SPPF)* is a representation that reduces the space used to represent multiple parse trees of an ambiguous string. Nodes are are named (x, j, i), when node x matches substring $a_{j+1} \ldots a_i$. Nodes which contain the same sub-

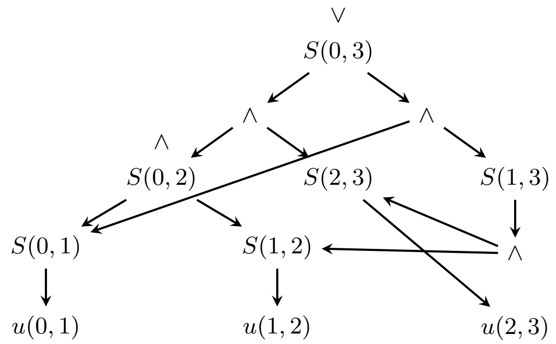

Figure 5. The SPPF representation of the two parse trees in Figure 3.

tree are shared and nodes which represent the same substring for nonterminals with different parse trees are packed.

5. Earley's Parser

Dr. Elizabeth Scott presented two algorithms that constructed SPPFs. The first algorithm decorates the Earley Sets with pointers as they are being constructed and subsequently walks through the sets using these pointers to create the SPPF. The second algorithm creates the SPPF as the Earley Sets are being constructed. We choose to discuss this second algorithm since the SPPF it produces is much cleaner than the one produced by the first algorithm. This is because some of the intermediate nodes created by the first algorithm are ignored by the second algorithm. The omission of these nodes not only decreases the size of the SPPF but also allows for a more efficient parser.

We will now discuss her algorithm which we will subsequently call the parser. At a high level, the parser loops through the characters in the input string, $a[1..n]$ and for each position i, it creates the Earley Set Σ_{i-1} and connects the leaf node representing a_i to the rest of the SPPF. This requires a modification the elements in the Earley Sets. Recall from Section 3 that each element in an Earley Set is a tuple (n_g, t), where n_g is a node in the GFG and t is the tag associated with that node. We simply extend this tuple to include one more field, (n_g, t, n_s), where n_s is the associated SPPF node. Recall from section 4 that $n_s = (x, j, i)$ when node x matches substring $a_{j+1} \dots a_i$. If an element in the Earley Set does not have an associated SPPF node, then n_s is simply null.

As we compute the ϵ-closure over the GFG, we maintain two auxilliary sets, Q and R. Q contains the scan nodes we encountered in the current iteration while R contains the other nodes. Intuitively R is a worklist that contains the nodes that still need to be processed in the current iteration and Q are the nodes that need to be processed to initialize the next Earley Set. Keep in mind that the whole point of an SPPF is to faciliate structure sharing, so when a node is desired we first check if it already exists. We maintain a cache, V, of nodes that are created in the current iteration for that purpose.

Special care must be taken for Earley elements of the form (n_g, i, n_s) when n_g is an exit node in Σ_i. This indicates that no scan edges were traversed between the start node and the exit node, meaning we have encountered an ϵ-production. Therefore the corresponding Earley element representing the call node must exist in Σ_i. The catch is that the Earley element representing the call node could be created after the the Earley element representing the exit node have already been processed. To this end, we maintain a set H in which tuples of the form (n_g, n_s) are added after $(n_g, i, n_s) \in \Sigma_i$ have been processed. Subsequently, when an Earley element representing a call node is processed, H is consulted.

We now present her algorithm:

Algorithm 1: EarleyParser

Input:
  A grammar $\Gamma = (N, T, P, S)$
  A string $a[1..n]$

Output:
  The corresponding Earley Sets and SPPF if G produces a

```
begin
Initialize Σ_0 ... Σ_n, R, Q', V = ∅ ;
Begin at (•S) in the GFG and compute the ϵ-closure ;
  Let n denote the reachable nodes that are not scan nodes ;
  Add (n, 0, null) to Σ_0 ;
  Let s denote the reachable nodes that are scan nodes ;
  Add (s, 0, null) to Q' ;
for 0 ≤ i ≤ n do
  Set R = Σ_i, Q = Q' ;
  Initialize H, Q' = ∅ ;
  Worklist(R, Q, H, V) ;
  Set V = ∅ ;
  Create a node v = (a_{i+1}, i, i + 1) ;
  while Q ≠ ∅ do
    Remove an element q = (n_g, t, n_s) from Q ;
    n_g is a scan node so there is an edge (n_g, n'_g) in the
    GFG ;
    y = MakeNode(n'_g, t, i + 1, n_s, v, V) ;
    Let n denote the reachable nodes that are not scan
    nodes ;
    Add (n'_g, t, y) to Σ_{i+1} ;
    Let s denote the reachable nodes that are scan nodes
    ;
    Add (n'_g, t, y) to Q' ;
Let s_e by the end node for the Start Symbol ;
if (s_e, 0, n_s) ∈ Σ_n then
  return n_s
else
  fail
end
```

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

We would like to thank Dr. Keshav Pingali for his guidance on this project. We would also like to thank Sepideh Maleki for helping us understand Scott's algorithm.

References

[1] Gianfranco Bilardi, and Keshav Pingali. Parsing with Pictures. UTCS Tech Reports, 2012. This is a full TECHREPORT entry.

[2] Elizabeth Scott. SPPF-Style Parsing From Earley Recognisers. *Electronic Notes in Theoretical Computer Science*, 203(53-67), 2008. This is a full ARTICLE entry.

Algorithm 2: Worklist

Input:

R, the nodes that still need to be processed in the current iteration

Q, the nodes that need to be processed to initialize the next Earley Set

H, the call nodes to epsilon productions

V, cache of nodes created during the current iteration

begin while $R \neq \emptyset$ **do**

- Remove an element $r = (n_g, t, n_s)$ from R ;
- **if** *r represents a call node for a nonterminal C* **then**
  - Compute the ϵ-closure ;
  - Let n denote the reachable nodes that are not scan nodes ;
  - Add $(n, i, null)$ to Σ_i and R if they are not in them already ;
  - Let s denote the reachable nodes that are scan nodes ;
  - Add $(s, i, null)$ to Q ;
  - Let n_e be the exit node corresponding to n_g ;
  - **if** $(n_e, n_s) \in H$ **then**
    - Let n_h be the return node corresponding to n_g ;
    - $y = MakeNode(n_h, t, i, n_s, null, V)$;
    - Let n denote the reachable nodes that are not scan nodes ;
    - Add (n, t, y) to Σ_i and R if they are not in them already ;
    - Let s denote the reachable nodes that are scan nodes ;
    - Add (s, t, y) to Q ;
- **if** *r represents an exit node for a nonterminal D* **then**
  - **if** *n_s is null* **then**
    - Create a node $v = (n_g, i, i)$ if it is not already in V ;
    - set $n_s = v$;
    - Make ϵ a child of n_s if it isn't already ;
  - **if** $t == i$ **then**
    - add (n_g, n_s) to H
  - **for** *$\sigma \in \Sigma_t$ representing call nodes, n_c, corresponding to n_g* **do**
    - Let $\sigma = (n_c, t', n'_s)$;
    - Let n_h be the return nodes corresponding to n_c ;
    - $y = MakeNode(n_h, t', i, n'_s, n_s, V)$;
    - Let n denote the reachable nodes that are not scan nodes ;
    - Add (n, t', y) to Σ_i and R if they are not in them already ;
    - Let s denote the reachable nodes that are scan nodes ;
    - Add (s, t', y) to Q ;

end

Algorithm 3: MakeNode

Input:

n_g, a GFG node

j, i substring indices where $j \leq i$

w, v, SPPF nodes

V, cache of nodes created during the current iteration

Output:

y, a new SPPF node

begin

if *n_g is an exit node of a nonterminal B* **then**

- set $s = B$

else

- set $s = n_g$

if *n_g is the second scan node and not an exit node* **then**

- set $y = v$

else

- Create a node $y = (s, j, i)$ if it is not already in V and add it to V ;
- **if** *w is null* **then**
  - make v a child of y if it isn't already
- **if** *w is not null* **then**
  - make w and v children of y if they aren't already

return y ;

end