Efficient Earley Parsing with Regular Right-hand Sides

Trevor Jim [1] Yitzhak Mandelbaum [2]

AT&T Labs Research

Abstract

We present a new variant of the Earley parsing algorithm capable of efficiently supporting context-free grammars with regular right hand-sides. We present the core state-machine driven algorithm, the translation of grammars into state machines, and the reconstruction algorithm. We also include a theoretical framework for presenting the algorithm and for evaluating optimizations. Finally, we evaluate the algorithm by testing its implementation.

Key words: Context-free grammars, Earley parsing, regular right sides, scannerless parsing, transducers, augmented transition networks

1 Introduction

This paper introduces a new parsing algorithm with three important features: it handles arbitrary context-free languages, it supports scannerless parsing, and it directly handles grammars with regular right sides.

These features all come into play when generating parsers for message formats used in network protocols such as HTTP, mail, and VoIP (SIP). These formats are defined by grammars in English-language specification documents called "Requests for Comments." The grammars are intended as documentation and therefore do not obey the restrictions typically imposed by parser generators used in the programming languages community. The grammars cannot be directly handled by existing parser generators, [3] so parsers are usually written by hand, an error-prone process that can lead to interoperability problems. Our algorithm can handle these grammars directly, without requir-

[1] Email: `trevor@research.att.com`
[2] Email: `yitzhak@research.att.com`
[3] There are a few tools which can take RFC grammars as input, but they do not generate correct parsers because of grammar ambiguities.

ing any hand-rewriting—an important consideration since the grammars are complex, ambiguous, and large, with hundreds of nonterminals.

These features are also useful in other areas, such as parsing programming languages. Many others have already pointed out the advantages of using abitrary context-free language parsers [9] and scannerless parsers [12] for programming languages, but regular right sides are less appreciated, and less supported by parser generators.

The principal advantage of regular right sides is that they are more concise and readable than their non-regular equivalents. This is demonstrated by the fact that grammars in documentation typically use regular right sides. This includes not only the Requests for Comments we have already mentioned, but also programming language documentation. For example, Python, Haskell, Ocaml, and even Pascal use some form of regular right side grammars in their official language specifications.

When languages are documented with a more restricted grammar formalism, this is often expressly so that their grammars can be used with a parser generator. The most well-known example is C, whose grammar is formulated to be compatible with YACC [6] (see Appendix B of Kernighan and Ritchie [8]). This is convenient for compiler writers, but perhaps less useful to the much larger community of C programmers.

Our parsers are based on Earley's parsing algorithm [5], so they can parse arbitrary context-free languages. They are scannerless, so they do not require syntax specifications to be split into separate lexing and parsing phases (although, lexers can be employed if desired). Finally, they handle regular right sides *directly*, without "desugaring" into a grammar without regular right sides.

Directly handling regular right sides is important for two reasons. First, desugaring introduces difficulties for the parser generator in relating the desugared grammar to the original for processing semantic actions, reporting errors, and for visualizing and debugging the parsing process. Second, and more importantly, desugaring results in less efficient parsers. Intuitively, this is because parsers must work harder when their grammars have more ambiguities. Regular expressions are highly ambiguous (e.g., $(x^*)^*$) and desugaring promotes these ambiguities into the grammar's nonterminals, that is, into parse trees. Our benchmarks show an overhead of 22–58% for desugaring.

Theoretically, our parsers are most closely related to the *augmented transition networks (ATNs)* of Woods [13], which also handle regular right sides directly, and which have previously been used for Earley parsing [2]. Our formalism uses a new kind of finite-state transducer which allows us to express some optimizations which significantly improve performance over ATNs. Some of these optimizations are similar to ones used by the Earley parsers of Aycock and Horspool [1], which do not directly handle regular right sides.

In this paper, we present and evaluate the new Earley parsing algorithm. In Section 2, we introduce a new *parsing transducer*, and present a non-

$$S = A \mid x^*y \qquad A = \epsilon \mid BAC \qquad B = x \qquad C = y$$

Fig. 1. Grammar for the language $x^n y^n \mid x^*y$; S is the start nonterminal.

deterministic parsing algorithm in which the transducer acts as the control for a pushdown automaton. We formalize the relationship between grammars and parsing transducers and specify sufficient conditions under which a given transducer will parse the language of a given grammar. In Section 3, we present an algorithm for constructing an appropriate transducer given a grammar as input. In Section 4, we present a deterministic, Earley-style, recognition algorithm based on parsing transducers and show that, for a given transducer, it recognizes the same language as the nondeterminisitic pushdown algorithm. We describe how to adapt the algorithm for parsing and present an algorithm for parse reconstruction in Section 5. We evaluate the parsing algorithm by comparing its performance to previous approaches in Section 6, and discuss related work in Section 7.

2 Foundations

Here we lay out the theoretical foundations of our parsing algorithm, omitting proofs due to space limitations.

2.1 Grammars and languages

A *grammar* G is a four-tuple $(\Sigma, \Delta, A_0, \mathcal{L})$ where

- Σ is a finite set of terminals;
- Δ is a finite set of nonterminals;
- $A_0 \in \Delta$ is the start nonterminal; and
- $\mathcal{L}$ is a family of regular languages over $(\Sigma \cup \Delta)$, indexed by Δ: $\mathcal{L}_{A_0}, \ldots$.

We use A, B, C to range over nonterminals, x, y, z to range over terminals, w to range over sequences of terminals, and W to range over sequences of terminals and nonterminals. ϵ is the empty sequence.

For every nonterminal A we define a language L_A by simultaneous induction:

$$\frac{w_0 A_1 w_1 \cdots A_n w_n \in \mathcal{L}_A, \quad w'_i \in L_{A_i} \text{ for all } i < n}{w_0 w'_1 w_1 \cdots w'_n w_n \in L_A} \quad (n \geq 0)$$

The language of G is defined to be the language of the start nonterminal A_0: $L_G = L_{A_0}$.

Figure 1 gives an example grammar for the language $x^n y^n \mid x^*y$, written as rules using regular expressions for the right-hand sides of the nonterminals.

$$\text{INPUT}\ \frac{s \to_x t}{(s,\alpha) \Rightarrow_x (t,\alpha)} \qquad \text{CALL}\ \frac{s \xrightarrow{\textbf{call}} t}{(s,\alpha) \Rightarrow_\epsilon (t, ts\alpha)}$$

$$\text{RESET}\ \frac{s \mapsto A, \quad t \to_A u}{(s,t\alpha) \Rightarrow_\epsilon (u,t\alpha)} \qquad \text{RETURN}\ \frac{s \mapsto A, \quad v \to_A u}{(s,tv\alpha) \Rightarrow_\epsilon (u,\alpha)}$$

Fig. 2. Evaluation of parsing transducers

2.2 Parsing transducers

Our parsers are closely related to the augmented transition networks (ATNs) of Woods [13]. ATNs are pushdown machines whose control is provided by a set of automata, one for each nonterminal of the grammar; the automaton for a nonterminal corresponds to its regular right side.

We use a generalization of this approach, replacing the automata with *transducers*, that is, automata with outputs. Transducers in general can have outputs on transitions and on final states, but it is sufficient for our purposes here to consider outputs on final states only.

A *parsing transducer* T is an 8-tuple $(\Sigma, \Delta, Q, A_0, q_0, \to, \xrightarrow{\textbf{call}}, \mapsto)$ where

- Σ is a finite set of terminals;
- Δ is a finite set of nonterminals;
- Q is a finite set of states;
- $A_0 \in \Delta$ is the start nonterminal;
- $q_0 \in Q$ is the initial state;
- $\to \subseteq Q \times (\Sigma \cup \Delta) \times Q$ is the transition relation;
- $\xrightarrow{\textbf{call}} \subseteq Q \times Q$ is the call relation; and
- $\mapsto \subseteq Q \times \Delta$ is the output relation from final states to nonterminals.

We use q, r, s, t, u, v to range over states and α, β, γ to range over sequences of states, and we write $q \to_W r$ if $(q, W, r) \in \to$. A *configuration* is a pair (q, α) where α acts as a stack that grows to the left. Figure 2 defines $\Rightarrow_w$, a single-step evaluation relation for configurations. The multi-step evaluation relation, $\Rightarrow_w^*$, is defined as usual. Notice that the multi-step evaluation relation defines a nondeterministic algorithm for parsing inputs with the parsing transducer.

Three of the four evaluation rules are (almost) ordinary: INPUT consumes a character of input; CALL nondeterministically starts a parse of a nonterminal at the current input position, recording some state on the stack; and RETURN detects that a nonterminal has parsed correctly (as indicated by reaching a state with that nonterminal as output) and pops the stack.

There is one twist: when a new parse is started, CALL pushes *both* the caller and the callee on the stack. RETURN uses the caller to find the next state ("return address") and pops both caller and callee from the stack. RESET

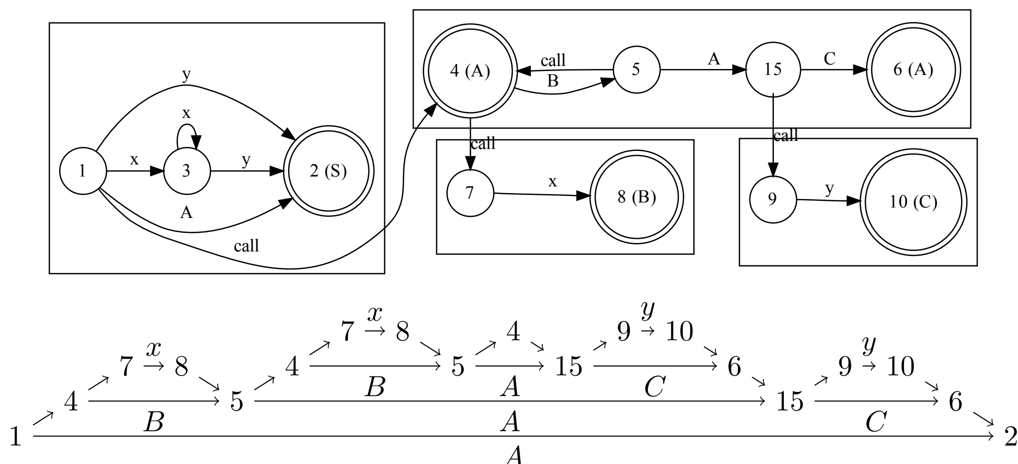


Fig. 3. A transducer for parsing $x^n y^n \mid x^* y$, and a transducer evaluation for $xxyy$.

uses the callee to find the next state, but does not pop the stack. As we will see in our discussion of Figure 4 and in Section 3, this provides more opportunities for left-factoring a grammar while building a transducer.

Finally, $L_A(q)$, *the language of* A *at* q, is $\{ w \mid (q,q) \Rightarrow_w^* (r,q),\ r \mapsto A \}$, and the language of the transducer $L_T = L_{A_0}(q_0)$. Notice that we start evaluation with q_0 on the stack; the assumption is that there is always a callee at the top of the stack, for use by the rule RESET.

Figure 3 gives a transducer for parsing the language of Figure 1, and gives a transducer evaluation in graphical form. This particular transducer is roughly equivalent to the ATN formulation of Woods. Woods uses a disjoint automaton for each right-hand side, and we have boxed these in the figure: the nodes 1, 4, 7, and 9 correspond to the automata for S, A, B, and C, respectively. In the evaluation, $\nearrow$-edges indicate CALL, $\searrow$-edges indicate RETURN, arrows labeled with nonterminals indicate a parse completed by the RETURN, and arrows labeled with terminals indicate INPUT. The rule RESET is not used in this evaluation.

A more efficient transducer for the same language is given in Figure 4. In this transducer, there are no longer disjoint state machines corresponding to the right-hand sides. For example, state 8 begins a simultaneous parse of both A and B, and state 1 begins a simultaneous parse of S, A, and B (in particular, the parses of S and B have been left-factored). The evaluation makes use of the RESET rule three times, indicated by the dotted arrow. Notice that the evaluation is shorter and the stack does not grow as high as in the first evaluation. In Section 3, we describe an algorithm for constructing optimized transducers like this one.

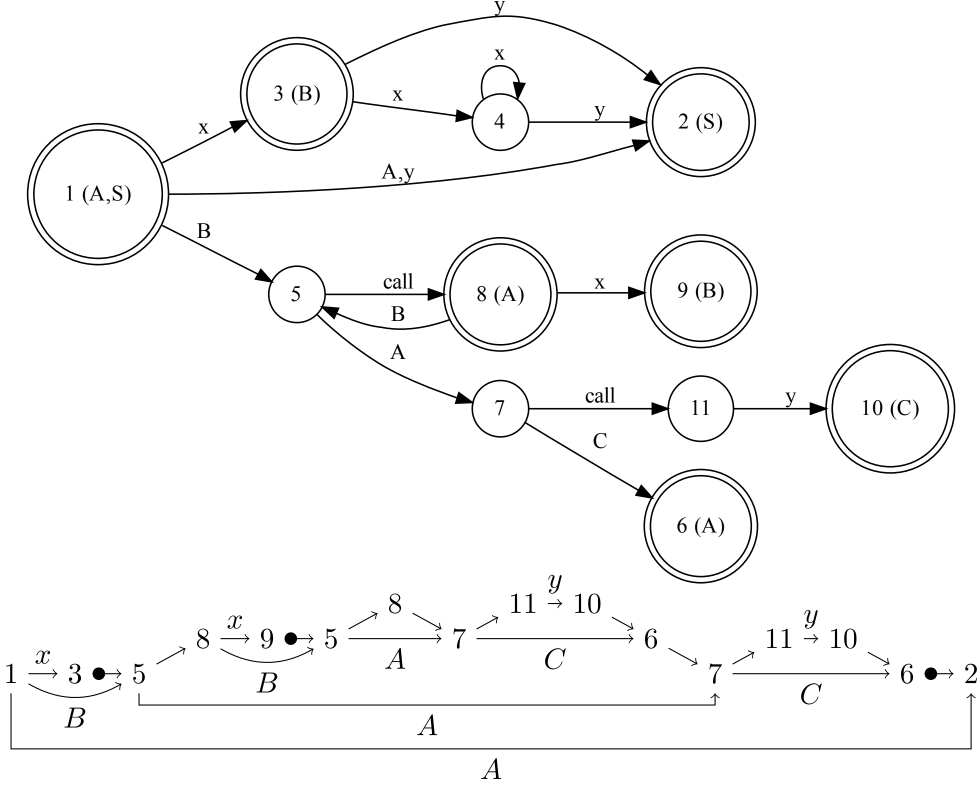


Fig. 4. An optimized transducer for parsing $x^n y^n \mid x^* y$, and an evaluation for $xxyy$.

2.3 Grammars and transducers

Our examples show that there can be multiple transducers capable of parsing a grammar. Here, we sketch conditions sufficient to ensure that a grammar and a transducer accept the same language. These conditions essentially say: *the transition relation $\rightarrow$ of the transducer induces the language of a right-hand side at callees.* An important consequence is that transformations like determinization and minimization which do not change the language induced by $\rightarrow$ can be used to optimize transducers.

We write $\rightarrow^*_W$ for the iterated composition of $\rightarrow$, and define $\mathcal{L}_A(q) = \{\, W \mid q \rightarrow^*_W r \wedge r \mapsto A \,\}$. We say q *is a callee* if $q = q_0$ or $\exists r.r \xrightarrow{\textbf{call}} q$. We say that a parsing transducer $T = (\Sigma, \Delta, Q, A_0, q_0, \rightarrow, \xrightarrow{\textbf{call}}, \mapsto)$ *is a transducer for* grammar $G = (\Sigma, \Delta, \mathcal{L}, A_0)$ if the following conditions hold.

(T0) $\mathcal{L}_{A_0}(q_0) = \mathcal{L}_{A_0}$.

(T1) If q is a callee and $q \rightarrow_A r$, then $\mathcal{L}_A(q) = \mathcal{L}_A$.

(T2) If $q \rightarrow_W r \rightarrow_A s$, then $r \xrightarrow{\textbf{call}} t$ for some t with $\mathcal{L}_A(t) = \mathcal{L}_A$.

(T3) If q is a callee, then $\mathcal{L}_A(q) \subseteq \mathcal{L}_A$ for all A.

Theorem 1 *If T is a transducer for G, then $L_T = L_G$.*

3 From Grammar to Transducer

There are many ways to build transducers for grammars, keeping in mind only that we need to satisfy conditions **T0–3**. Constructing a transducer like the one in Figure 3 that corresponds to a Woods ATN is easy: use a standard method to convert each grammar right side into a deterministic finite automaton, mark final states as outputting their corresponding nonterminals, and add **call**-edges from each state with a nonterminal out-transition to the initial state of the corresponding nonterminal's automaton.

Building an optimized transducer like the one in Figure 4 is only a little more involved. The idea is to avoid consecutive sequences of calls, in essence replacing any $s \rightarrow_{\mathbf{call}} t \rightarrow_{\mathbf{call}} u$ with $s \rightarrow_{\mathbf{call}} t \rightarrow_{\epsilon} u$. Our algorithm is given in Figure 5. Its input is a grammar defined by (syntactic) regular right sides, and its output is a transducer with ϵ-transitions. We obtain a parsing transducer for the grammar (satisfying conditions **T0–3**) by applying standard determinization and minimization transformations.

The key procedure is $conv(r)$, which produces a triple (s, F, E) of states. The state s is the initial state for r, and F and E are final states. The construction ensures that all paths from s to F contain a non-**call**, non-ϵ transition, and that all paths from s to E contain only ϵ-transitions. This invariant is used in the cases $r = (r_2 r_3)$ and $r = (r_2^*)$ to ensure condition **T2**.

The function $thompson'$ is Thompson's algorithm for converting a regular expression to an NFA [11], extended with a case for a nonterminal A: it produces both an A-transition and a **call**-transition. In contrast, $conv(A)$ uses an ϵ-transition instead of a **call**-transition. Note that $thompson'$ is used in the cases $r = (r_2 r_3)$ and $r = (r_2^*)$ rather than $conv$ exactly to ensure **T2**.

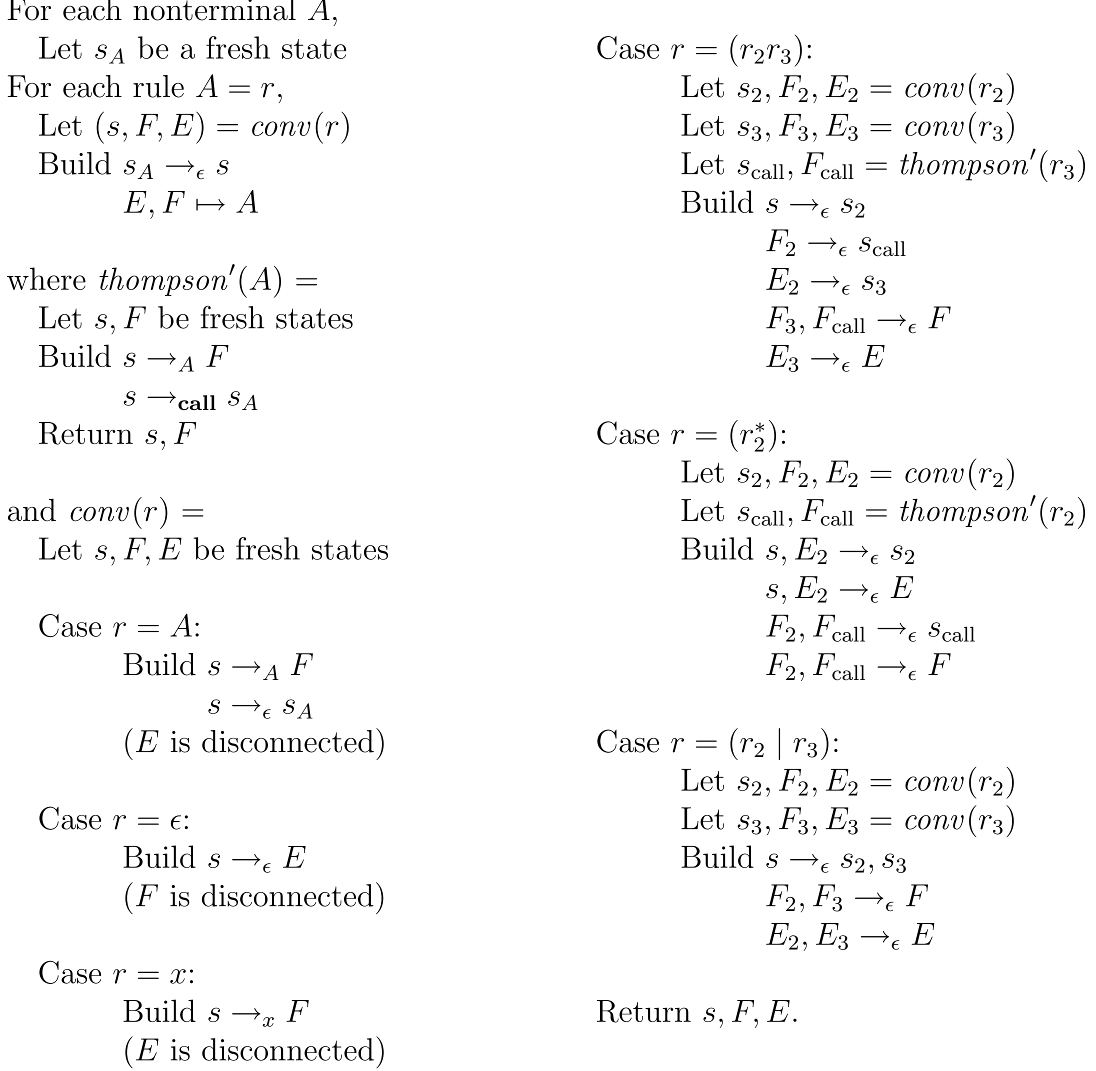
4 Earley Parsing for Transducers

Earley parsing is a top-down parsing method that can parse any context-free language [5]. It performs a breadth-first search of all possible leftmost derivations of the input string. Earley parsing has an upper bound of $O(n^3)$, where n is the size of the input, and Earley showed that his algorithm works in $O(n^2)$ time for unambiguous grammars, and in linear time for a large class of grammars. Our version of Earley retains the $O(n^3)$ upper bound.

In the rest of this section we show how Earley parsing can be applied to our parsing transducers.

4.1 The Earley Sets

For an input $x_1 x_2 \ldots x_n$, we define the *Earley sets* to be the least sets satisfying the rules of Figure 6. Each element $(q, i) \in E_j$ is called an *Earley item*



For each nonterminal A,
  Let s_A be a fresh state
For each rule $A = r$,
  Let $(s, F, E) = conv(r)$
  Build $s_A \to_\epsilon s$
    $E, F \mapsto A$

where $thompson'(A) =$
  Let s, F be fresh states
  Build $s \to_A F$
    $s \to_{\textbf{call}} s_A$
  Return s, F

and $conv(r) =$
  Let s, F, E be fresh states

  Case $r = A$:
    Build $s \to_A F$
      $s \to_\epsilon s_A$
    (E is disconnected)

  Case $r = \epsilon$:
    Build $s \to_\epsilon E$
    (F is disconnected)

  Case $r = x$:
    Build $s \to_x F$
    (E is disconnected)

  Case $r = (r_2 r_3)$:
    Let $s_2, F_2, E_2 = conv(r_2)$
    Let $s_3, F_3, E_3 = conv(r_3)$
    Let $s_{\text{call}}, F_{\text{call}} = thompson'(r_3)$
    Build $s \to_\epsilon s_2$
      $F_2 \to_\epsilon s_{\text{call}}$
      $E_2 \to_\epsilon s_3$
      $F_3, F_{\text{call}} \to_\epsilon F$
      $E_3 \to_\epsilon E$

  Case $r = (r_2^*)$:
    Let $s_2, F_2, E_2 = conv(r_2)$
    Let $s_{\text{call}}, F_{\text{call}} = thompson'(r_2)$
    Build $s, E_2 \to_\epsilon s_2$
      $s, E_2 \to_\epsilon E$
      $F_2, F_{\text{call}} \to_\epsilon s_{\text{call}}$
      $F_2, F_{\text{call}} \to_\epsilon F$

  Case $r = (r_2 \mid r_3)$:
    Let $s_2, F_2, E_2 = conv(r_2)$
    Let $s_3, F_3, E_3 = conv(r_3)$
    Build $s \to_\epsilon s_2, s_3$
      $F_2, F_3 \to_\epsilon F$
      $E_2, E_3 \to_\epsilon E$

  Return s, F, E.

Fig. 5. Converting a regular right side grammar to a transducer with ϵ-transitions as part of transducer construction. Note that we only show the nonterminal case of the *thompson'* function.

and represents a parse which started at input position i and has progressed to position j, resulting in transducer state q. Thus, rule INIT says that we start parsing in state q_0 at input position 0, SCAN corresponds to INPUT, and PREDICT corresponds to CALL. Rule COMPLETE corresponds to both RETURN and RESET, where the return address for a parse starting at i is found from the set E_i.

Technically, the Earley set construction defines a *recognizer* that accepts an input if $(q, 0) \in E_n$ where $q \mapsto A_0$. Building a parse tree for a recognized input requires more work, as we discuss in Section 5.

Theorem 2 *If T is a transducer for G, then w is accepted by the Earley sets for T iff $w \in L_G$.*

$$\text{INIT} \qquad (q_0, 0) \in E_0$$

$$\text{SCAN} \qquad \frac{(t,i) \in E_{j-1} \quad t \to_{x_j} s}{(s,i) \in E_j}$$

$$\text{PREDICT} \qquad \frac{(t,i) \in E_j \quad t \xrightarrow{\textbf{call}} s}{(s,j) \in E_j}$$

$$\text{COMPLETE} \qquad \frac{(t,k) \in E_j \quad t \mapsto A \quad (u,i) \in E_k \quad u \to_A s}{(s,i) \in E_j}$$

Fig. 6. The Earley sets

4.2 Building the Earley Sets

The Earley sets can be built by first "seeding" E_0 by rule INIT, then processing E_0, E_1, ..., E_n in sequence as follows:

(i) For the current set E_j, apply the rules PREDICT and COMPLETE until no more items can be added.

(ii) Use SCAN to seed E_{j+1}.

It is well known that this algorithm is inefficient when the grammar has nullable nonterminals (A is nullable if $\epsilon \in L_A$). If a nullable A is called in E_j, it will also return to E_j, having consumed no terminals. This means that COMPLETE will be invoked with $j = k$. Unfortunately, COMPLETE needs to consider the *full* set E_k to look for A-transitions, and if $j = k$ then E_k will in fact be the set under construction. Nullable nonterminals thus require extra processing.

For this reason, our algorithm replaces the rule COMPLETE with the rules in Figure 7. The rule NNCOMPLETE is just the special case of COMPLETE for a non-null A. The remaining rules handle null completion. NCALLER combines PREDICT with an immediate COMPLETE to the caller, and NCALLEE combines PREDICT with an immediate COMPLETE to the callee. NINIT does null completion for the initial Earley item, $(q_0, 0) \in E_0$; recall that q_0 is a special case of a callee for our parsing transducers, and so NINIT is analogous to NCALLEE.

The new rules define exactly the same Earley sets as the original rules, provided the transducer satisfies **T2** and the following condition:

(P1) If t is a callee for A (*i.e.*, $\mathcal{L}_A(t) = \mathcal{L}_A$) and A is nullable, then $t \mapsto A$.

To see this, consider that when any (t,i) with $t \to_A$ for nullable A is added to E_j by scanning or completion, **T2** says there will be a **call** from t, and **P1** ensures that NCALLER will perform the null completion on the transition $t \to_A$. When (t,i) is added by prediction, **P1** ensures that NCALLEE applies.

$$\text{NNCOMPLETE}\ \frac{(t,k) \in E_j \quad t \mapsto A \quad (u,i) \in E_k \quad u \to_A s}{(s,i) \in E_j}\ (j \neq k)$$

$$\text{NCALLER}\ \frac{(t,i) \in E_j \quad t \xrightarrow{\textbf{call}} s \mapsto A \quad t \to_A u}{(u,i) \in E_j}$$

$$\text{NCALLEE}\ \frac{(t,i) \in E_j \quad t \xrightarrow{\textbf{call}} s \mapsto A \quad s \to_A u}{(u,j) \in E_j}$$

$$\text{NINIT}\ \frac{q_0 \mapsto A \quad q_0 \to_A r}{(r,0) \in E_0}$$

Fig. 7. Rules for handling nullable symbols

The remaining case is for the initial item $(q_0, 0) \in E_0$, and here **P1** ensures that NINIT applies.

Some transducers do not satisfy **P1**, e.g., any transducer for the grammar

$$S = A, \quad A = \epsilon.$$

Here S is nullable, so by **P1** we want $q_0 \mapsto S$; but then **T0** implies that $\epsilon \in \mathcal{L}_S$, which is false. This is easily handled, however: any grammar can be transformed into a grammar that recognizes the same language and whose transducers satisfy **P1** by replacing $A = r$ with $A = (r \mid \epsilon)$ for all nullable A.

For efficiency, our algorithm applies all of the rules PREDICT, NCALLER, and NCALLEE together whenever considering an item $(t,i) \in E_j$ such that $t \xrightarrow{\textbf{call}} s$ in the closure phase.

5 Parse Reconstruction

The algorithm of Section 4 is a recognizer: it accepts exactly the inputs that parse according to the grammar, but it does not return a parse tree or parse forest for the input. Here we extend the algorithm so that trees and forests can be reconstructed for an input.

The main idea is to modify the recognizer to remember some information as it applies rules like SCAN. The stored information takes the form of a graph whose nodes are labeled with Earley items, and whose edges indicate why a node is added to an Earley set. The resulting graph embodies a parse forest, and we show how to extract parse trees from it.

Our method can be seen as an extension of the method of Aycock and Horspool [1] to handle regular right sides. It is also similar to the work of Scott [10], who showed how to efficiently modify the traditional Earley algorithm to construct a *shared packed parse forest* (SPPF). We discuss forest reconstruction in Section 5.3.

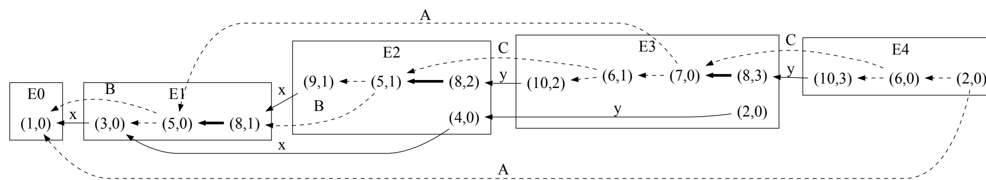


Fig. 8. The Earley graph for parsing $xxyy$.

5.1 Earley graphs

The first step in reconstruction comes during the parsing process itself. We extend the algorithm of Section 4 to construct an *(Earley) parse graph* during parsing. Each node of the graph corresponds to an Earley item in an Earley set. Directed edges from a node describe how rules were used during parsing to add the Earley item to the Earley set.

We explain Earley graphs by example. Figure 8 shows the Earley graph of the transducer of Figure 4 evaluated on the input $xxyy$. There are several kinds of edges in the graph.

- Edges labeled with terminals correspond to use of the rule SCAN. For example, the item (3,0) is added to E_1 by scanning x starting from item (1,0) in E_0.
- Bold edges correspond to the use of prediction. For example, the item (8,1) is added to E_1 by prediction starting at item (5,0) in E_1.
- Dashed edges correspond to completion. The dashed edges come in pairs—they are multiedges. One edge is labeled with the nonterminal that has been completed, and it points at the node which initiated the parse of the nonterminal. The other edge points to the node which completed the parse of the nonterminal. For example, the item (5,0) is added to E_1 as a result of completing a parse of B that was initiated by item (1,0) in E_0, and completed by item (3,0) in E_1.

The second kind of edge, for prediction, is not used by Aycock and Horspool or by Scott. It is necessary in our case to handle a subtle detail that we will discuss in Section 5.3.

Earley graphs encode sets of parse trees. For example, in the graph of Figure 8, two parse trees for S are evident. One is rooted at item (2,0) in E_3, and it indicates a complete parse of the prefix xxy by three scans. The second tree is rooted at item (2,0) in E_4, and it corresponds to the parse tree for the full input given in Figure 4.

5.2 Reconstructing a parse tree

In Figure 9, we sketch the algorithm for nondeterministically reconstructing a parse tree for the input from the Earley graph. The algorithm is

```
fun mk_parse_tree sym item_complete =
  fun complete_children item children =
    if empty (links item) then
      children
    else
    let l = CHOOSE from links item in
    if is_nonterminal l then
      let child = mk_parse_tree (sym_sub l) (item_sub l)  in
      complete_children (item_pred l) (child::children)
    else
      complete_children (item_pred l) children
  in
  let children = complete_children item_complete [] in
  ( sym, item_complete.i, item_complete.j, children )
```

Fig. 9. Tree Reconstruction

defined in terms of two mutually recursive functions, `mk_parse_tree` and `complete_children`. The function `mk_parse_tree` takes two arguments: the nonterminal for which to reconstruct the parse tree (`sym`); and the node at which the nonterminal was completed (`item_complete`). It returns a parse tree consisting of the nonterminal, its span in the input, and its children (if any). Function `complete_children` takes a node and a list of children and extends the list to be a complete reconstruction of the nonterminal associated with the argument node.

`mk_parse_tree` reconstructs the children of the parse tree from right to left. It starts with an empty list and the rightmost item of the completed nonterminal (`item_complete`). It then calls `complete_children` to (recursively) traverse predecessor links until reaching an item with no links, which must be a callee of some call. While traversing links, `complete_children` reconstructs subtrees when nonterminal links are encountered. Subtree reconstruction is accomplished with a recursive call to `mk_parse_tree`. Some items may have been the target of more than one transition, resulting in multiple links. In this case, the algorithm nondeterministically chooses a single link to follow with the abstract `CHOOSE` function.

5.3 Reconstructing a parse forest

We have argued that the Earley graph embodies all of the parse trees for an input, and have just described how to reconstruct a parse tree from the Earley graph. However, we may wish to have a more accessible representation of the parse trees than the Earley graph: a parse forest. We have implemented a parse forest reconstruction algorithm, but do not have space to describe it here in full. Instead, we will discuss some subtleties that arise due to our use of parsing transducers.

Our transducer construction is designed to reduce the number of items

Grammar	Input Size	Execution Time			Overhead	
		Regular	Desugared	ATN	Desugared	ATN
IMAP	330 KB	1.09	1.72	1.61	58%	48%
OCaml	228 KB	1.15	1.40	5.77	22%	400%

Table 1
Average parse times and overheads. Each average was derived from 10 trials.

within Earley sets, by combining states (*i.e.*, LR(0) items) which always appear together. However, such combinations can potentially lead to a single Earley state becoming part of multiple transducer states. In particular, this can occur for callee states. For example, consider the following grammar:

$$S = x(A \mid yB) \qquad A = y(B \mid C) \qquad B = www \qquad C = z$$

The start state of B will appear in two different transducer states—one with the start state of C and one without. As is common in subset construction, multiple paths of this sort can converge into a single path within the automaton. Correspondingly, there can be multiple paths within the Earley sets which converge to a single path. Usually, convergence indicates an ambiguity at the point of convergence. However, in this case it does not (it does, however, indicate an ambiguity at some higher level in the call stack). Therefore, when reconstructing the parse forest, we must take care not to treat such convergences as indicative of multiple parse paths. Note that Scott's algorithm for reconstructing parse forests does not encounter this problem because it was designed for the traditional Earley algorithm, in which LR(0) items are never grouped together.

In order to filter such erroneous parse trees, we need to filter paths by their call source. Therefore, during parsing, we must track the set of callers for each state. Then, during reconstruction we provide the reconstruction function with the item responsible for calling the nonterminal under reconstruction. Next, we filter links whose caller sets do not include the specified caller. [4] Finally, we specify the appropriate caller to the reconstruction function by providing the predecessor item of the nonterminal link.

6 Evaluation

We have implemented several versions of our algorithm and compare them in Table 1. "Regular" is the recognition algorithm as described in this paper. "Desugared" is the same algorithm, but using a grammar obtained desugaring regular right sides from the original grammar. We believe that this should

[4] In order to filter links in this way, the caller sets of each link must be included in the Earley graph. In our implementation, we only include caller sets for callees and we filter paths only upon completion (by rejecting paths with mismatched callers).

approximate the PEP algorithm of Aycock and Horspool and show the performance advantage of directly handling regular right sides. Finally, "ATN" is a version of the algorithm that uses a transducer that closely corresponds to Woods' augmented transition networks (see Section 3). This should show the performance advantage of using our optimized parsing transducers.

We tested our algorithm on two grammars, whose styles are markedly different: IMAP, the popular mail protocol [4]; and the OCaml programming language. The IMAP grammar was extracted automatically from several RFCs, and it is written in a scannerless style. We adapted the OCaml grammar from the Lex and Yacc specifications available in the OCaml compiler source distribution (3.09.3). The IMAP input is a 330KB trace generated by concatenating a smaller trace (3.3KB) multiple times. The OCaml input is ∼50 OCaml source files, totaling 228KB; the largest file is 34KB. We ran all of the experiments on MacBook Pro with a 2.33 GHz Intel Core 2 Duo processor and 2 GB of RAM.

Table 1 gives our results. The first three columns of numbers give the execution times of the three algorithms. The final two columns give the overhead of the the Desugared and ATN variants, relative to the Regular variant. Note that for both grammars, the other methods show overheads of 22–400%, demonstrating a clear advantage for our algorithm. Second, the relationships between the algorithms differs for IMAP and OCaml, which is not surprising considering the difference in the grammar styles.

7 Related Work

Earley's original algorithm was one of the first parsers for general context-free languages [5]. It does not directly handle regular right sides.

Woods' augmented transition networks [13] are a particularly elegant parsing formalism, which seems little-known in the programming language community. They directly handle regular right sides, and have been used in Earley parsing [2]. Woods also considered ATNs based on transducers [14], though not in the context of Earley parsing. Our transducers are designed specifically to be more efficient than ATNs for Earley parsing.

Aycock and Horspool's Earley parsing algorithm [1] uses automata similar to the transducers we construct in Section 3 (though they arrive there in a very different way); in particular, they satisfy a property like our **T2**. Their parser is not scannerless and does not directly handle regular right sides.

We are not aware of another parsing algorithm that handles general context-free grammars with regular right sides without desugaring. There has been some work on regular right sides for LR(1) subsets [7,3].

Scott shows how to construct SPPFs using Earley parsing [10]. Her algorithm is designed for the traditional Earley algorithm; it would be interesting to study how to use her binarised SPPFs for grammars with regular right sides.

References

[1] John Aycock and R. Nigel Horspool. Practical Earley parsing. *Computer Journal*, 45(6):620–630, 2002.

[2] S. M. Chou and K. S. Fu. Transition networks for pattern recognition. Technical Report TR–EE–75–39, School for Electrical Engineering, Purdue University, West Lafayette, IN, 1975.

[3] Compiler Resources, Inc. Yacc++ and the language objects library. http://world.std.com/~compres/.

[4] M. Crispin. Internet message access protocol — version 4rev1. http://www.ietf.org/rfc/rfc3501.txt, March 2003.

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

[6] S. C. Johnson. Yacc: Yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[7] Sönke Kannapinn. *Reconstructing LR Theory to Eliminate Redundance, with an Application to the Construction of ELR Parsers.* PhD thesis, Technical University of Berlin, 2001.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, Inc., 1988.

[9] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *CC 2004: Compiler Construction, 13th International Conference*, 2004.

[10] Elizabeth Scott. SPPF-style parsing from Earley recognisers. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 53–67. Elsevier, 2008.

[11] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[12] M.G.D. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, April 2002.

[13] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606, 1970.

[14] W. A. Woods. Cascaded ATN grammars. *American Journal of Computational Linguistics*, 6(1):1–12, 1980.