Practical Earley Parsing

JOHN AYCOCK[1] AND R. NIGEL HORSPOOL[2]

[1]*Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada T2N 1N4*
[2]*Department of Computer Science, University of Victoria, Victoria, BC, Canada V8W 3P6*
Email: aycock@cpsc.ucalgary.ca

Earley's parsing algorithm is a general algorithm, able to handle any context-free grammar. As with most parsing algorithms, however, the presence of grammar rules having empty right-hand sides complicates matters. By analyzing why Earley's algorithm struggles with these grammar rules, we have devised a simple solution to the problem. Our empty-rule solution leads to a new type of finite automaton expressly suited for use in Earley parsers and to a new statement of Earley's algorithm. We show that this new form of Earley parser is much more time efficient in practice than the original.

Received 5 November 2001; revised 17 April 2002

1. INTRODUCTION

Earley's parsing algorithm [1, 2] is a general algorithm, capable of parsing according to *any* context-free grammar. (In comparison, the LALR(1) parsing algorithm used in Yacc [3] is limited to a subset of unambiguous context-free grammars.) General parsing algorithms like Earley parsing allow unfettered expression of ambiguous grammar constructs, such as those used in C, C++ [4], software reengineering [5], Graham/Glanville code generation [6] and natural language processing.

Like most parsing algorithms, Earley parsers suffer from additional complications when handling grammars containing ϵ-rules, i.e. rules of the form $A \rightarrow \epsilon$ which crop up frequently in grammars of practical interest. There are two published solutions to this problem when it arises in an Earley parser, each with substantial drawbacks. As we discuss later, one solution wastes effort by repeatedly iterating over a queue of work items; the other involves extra run-time overhead and couples logically distinct parts of Earley's algorithm.

This led us to study the nature of the interaction between Earley parsing and ϵ-rules more closely, and arrive at a straightforward remedy to the problem. We use this result to create a new type of automaton customized for use in an Earley parser, leading to a new, efficient form of Earley's algorithm.

We have organized the paper as follows. We introduce Earley parsing in Section 2 and describe the problem that ϵ-rules cause in Section 3. Section 4 presents our solution to the ϵ-rule problem, followed by a proof of correctness in Section 5. In Sections 6 and 7, we show how to construct our new automaton and how Earley's algorithm can be restated to take advantage of this new automaton. Section 8 discusses how derivations of the input may be reconstructed. Finally, some empirical results in Section 9 demonstrate the efficacy of our approach and why it makes Earley parsing practical.

2. EARLEY PARSING

We assume familiarity with standard grammar notation [7].

Earley parsers operate by constructing a sequence of sets, sometimes called Earley sets. Given an input $x_1 x_2 \ldots x_n$, the parser builds $n+1$ sets: an initial set S_0 and one set S_i for each input symbol x_i. Elements of these sets are referred to as (Earley) items, which consist of three parts: a grammar rule, a position in the right-hand side of the rule indicating how much of that rule has been seen and a pointer to an earlier Earley set. Typically Earley items are written as

$$[A \rightarrow \alpha \bullet \beta, j]$$

where the position in the rule's right-hand side is denoted by a dot ($\bullet$) and j is a pointer to set S_j.

An Earley set S_i is computed from an initial set of Earley items in S_i, and S_{i+1} is initialized, by applying the following three steps to the items in S_i until no more can be added.

SCANNER. If $[A \rightarrow \ldots \bullet a \ldots, j]$ is in S_i and $a = x_{i+1}$, add $[A \rightarrow \ldots a \bullet \ldots, j]$ to S_{i+1}.

PREDICTOR. If $[A \rightarrow \ldots \bullet B \ldots, j]$ is in S_i, add $[B \rightarrow \bullet\alpha, i]$ to S_i for all rules $B \rightarrow \alpha$.

COMPLETER. If $[A \rightarrow \ldots \bullet, j]$ is in S_i, add $[B \rightarrow \ldots A \bullet \ldots, k]$ to S_i for all items $[B \rightarrow \ldots \bullet A \ldots, k]$ in S_j.

An item is added to a set only if it is not in the set already. The initial set S_0 contains the item $[S' \rightarrow \bullet S, 0]$ to begin with—we assume the grammar is augmented with a new start rule $S' \rightarrow S$—and the final set must contain $[S' \rightarrow S \bullet, 0]$ for the input to be accepted. Figure 1 shows an example of Earley parser operation.

We have not used lookahead in this description of Earley parsing, for two reasons. First, it is easier to reason about Earley's algorithm without lookahead. Second, it is not clear what role lookahead should play in Earley's algorithm.

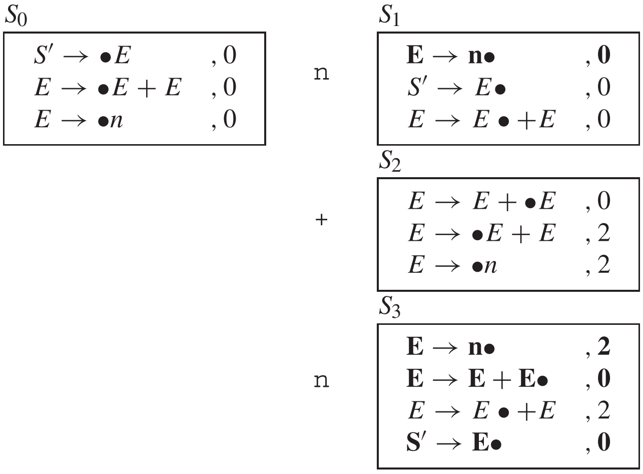
S_0
$S' \rightarrow \bullet E$, 0
$E \rightarrow \bullet E + E$, 0
$E \rightarrow \bullet n$, 0
n
S_1
$\mathbf{E \rightarrow n\bullet}$, **0**
$S' \rightarrow E\bullet$, 0
$E \rightarrow E \bullet + E$, 0
S_2
+
$E \rightarrow E + \bullet E$, 0
$E \rightarrow \bullet E + E$, 2
$E \rightarrow \bullet n$, 2
S_3
n
$\mathbf{E \rightarrow n\bullet}$, **2**
$\mathbf{E \rightarrow E + E\bullet}$, **0**
$E \rightarrow E \bullet + E$, 2
$\mathbf{S' \rightarrow E\bullet}$, **0**

FIGURE 1. Earley sets for the grammar $E \rightarrow E + E \mid n$ and the input `n + n`. Items in bold are ones which correspond to the input's derivation.

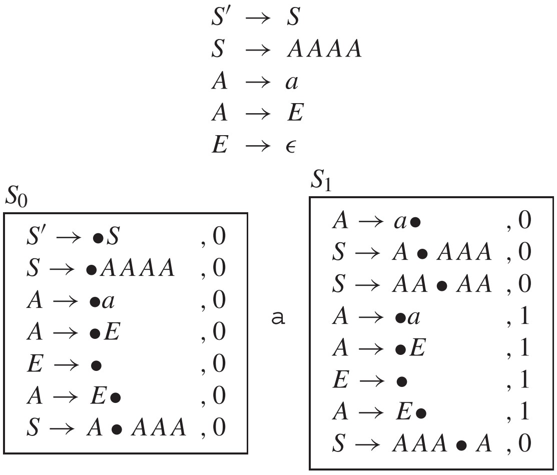
$S' \rightarrow S$
$S \rightarrow AAAA$
$A \rightarrow a$
$A \rightarrow E$
$E \rightarrow \epsilon$
S_0
$S' \rightarrow \bullet S$, 0
$S \rightarrow \bullet AAAA$, 0
$A \rightarrow \bullet a$, 0
$A \rightarrow \bullet E$, 0
$E \rightarrow \bullet$, 0
$A \rightarrow E\bullet$, 0
$S \rightarrow A \bullet AAA$, 0
a
S_1
$A \rightarrow a\bullet$, 0
$S \rightarrow A \bullet AAA$, 0
$S \rightarrow AA \bullet AA$, 0
$A \rightarrow \bullet a$, 1
$A \rightarrow \bullet E$, 1
$E \rightarrow \bullet$, 1
$A \rightarrow E\bullet$, 1
$S \rightarrow AAA \bullet A$, 0

FIGURE 2. An unadulterated Earley parser, representing sets using lists, rejects the valid input `a`. Missing items in S_0 sound the death knell for this parse.

Earley recommended using lookahead for the COMPLETER step [2]; it was later shown that a better approach was to use lookahead for the PREDICTOR step [8]; later it was shown that prediction lookahead was of questionable value in an Earley parser which uses finite automata [9] as ours does.

In terms of implementation, the Earley sets are built in increasing order as the input is read. Also, each set is typically represented as a list of items, as suggested by Earley [1, 2]. This list representation of a set is particularly convenient, because the list of items acts as a 'work queue' when building the set: items are examined in order, applying SCANNER, PREDICTOR and COMPLETER as necessary; items added to the set are appended onto the end of the list.

3. THE PROBLEM OF ϵ

At any given point i in the parse, we have two partially-constructed sets. SCANNER may add items to S_{i+1} and S_i may have items added to it by PREDICTOR and COMPLETER. It is this latter possibility, adding items to S_i while representing sets as lists, which causes grief with ϵ-rules.

When COMPLETER processes an item $[A \rightarrow \bullet, j]$ which corresponds to the ϵ-rule $A \rightarrow \epsilon$, it must look through S_j for items with the dot before an A. Unfortunately, for ϵ-rule items, j is always equal to i—COMPLETER is thus looking through the partially-constructed set S_i.[3] Since implementations process items in S_i in order, if an item $[B \rightarrow \ldots \bullet A \ldots, k]$ is added to S_i after COMPLETER has processed $[A \rightarrow \bullet, j]$, COMPLETER will never add $[B \rightarrow \ldots A \bullet \ldots, k]$ to S_i. In turn, items resulting directly and indirectly from $[B \rightarrow \ldots A \bullet \ldots, k]$ will be omitted too. This effectively prunes potential derivation paths, which can cause correct input to be rejected. Figure 2 gives an example of this happening.

Two methods of handling this problem have been proposed. Grune and Jacobs aptly summarize one approach:

> 'The easiest way to handle this mare's nest is to stay calm and keep running the Predictor and Completer in turn until neither has anything more to add.' [10, p. 159]

Aho and Ullman [11] specify this method in their presentation of Earley parsing and it is used by ACCENT [12], a compiler–compiler which generates Earley parsers.

The other approach was suggested by Earley [1, 2]. He proposed having COMPLETER note that the dot needed to be moved over A, then looking for this whenever future items were added to S_i. For efficiency's sake, the collection of non-terminals to watch for should be stored in a data structure which allows fast access. We used this method initially for the Earley parser in the SPARK toolkit [13].

In our opinion, neither approach is very satisfactory. Repeatedly processing S_i, or parts thereof, involves a lot of activity for little gain; Earley's solution requires an extra, dynamically-updated data structure and the unnatural mating of COMPLETER with the addition of items. Ideally, we want a solution which retains the elegance of Earley's algorithm, only processes items in S_i once and has no run-time overhead from updating a data structure.

4. AN 'IDEAL' SOLUTION

Our solution involves a simple modification to PREDICTOR, based on the idea of nullability. A non-terminal A is said to be nullable if $A \Rightarrow^* \epsilon$; terminal symbols, of course, can never be nullable. The nullability of non-terminals in a grammar may be easily precomputed using well-known techniques [14, 15]. Using this notion, our PREDICTOR can be stated as follows (our modification is in bold):

> If $[A \rightarrow \ldots \bullet B \ldots, j]$ is in S_i, add $[B \rightarrow \bullet\alpha, i]$ to S_i for all rules $B \rightarrow \alpha$. **If B is nullable, also add $[\mathbf{A} \rightarrow \ldots \mathbf{B} \bullet \ldots, \mathbf{j}]$ to S_i.**

[3] $j = i$ for ϵ-rule items because they can only be added to an Earley set by PREDICTOR, which always bestows added items with the parent pointer i.

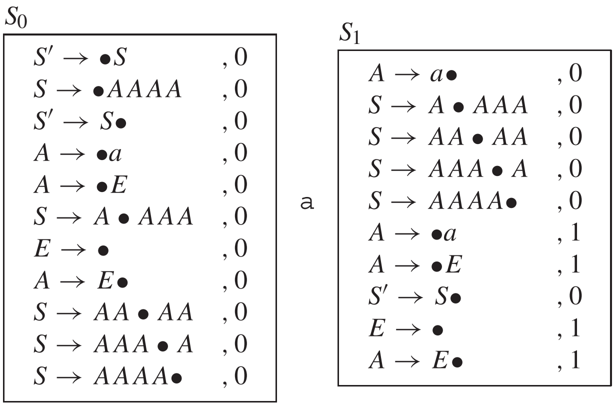

FIGURE 3. An Earley parser accepts the input a, using our modification to PREDICTOR.

In other words, we eagerly move the dot over a non-terminal if that non-terminal can derive ϵ and effectively 'disappear'. Using the grammar from the ill-fated Figure 2, Figure 3 demonstrates how our modified PREDICTOR fixes the problem.

5. PROOF OF CORRECTNESS

Our solution is correct in the sense that it produces exactly the same items in an Earley set S_i as would Earley's original algorithm as described in Section 2 (where an Earley set may be iterated over until no new items are added to S_i or S_{i+1}). In the following proof, we write S_i and S'_i to denote the contents of Earley set S_i as computed by Earley's method and our method, respectively. Earley's SCANNER, PREDICTOR and COMPLETER steps are denoted by E_S, E_P, and E_C; ours are denoted by E'_S, E'_P, and E'_C.

We begin with some results which will be used later.

LEMMA 5.1. *Let I be the Earley item $[A \to \alpha\bullet, i]$. If $I \in S_i$, then $\alpha \Rightarrow^* \epsilon$.*

Proof. There are two cases, depending on the length of α.

Case 1. $|\alpha| = 0$. The lemma is trivially true, because α must be ϵ.

Case 2. $|\alpha| > 0$. The key observation here is that the parent pointer of an Earley item indicates the Earley set where the item first appeared. In other words, the item $[A \to \bullet\alpha, i]$ must also be present in S_i, and because both it and I are in S_i, it means that α must have made its debut *and* been recognized without consuming any terminal symbols.

Therefore $\alpha \Rightarrow^* \epsilon$. □

LEMMA 5.2. *If $S_0 = S'_0$, $S_1 = S'_1, \ldots, S_{i-1} = S'_{i-1}$, then $S_i \subseteq S'_i$.*

Proof. Assume that there is some Earley item $I \in S_i$ such that $I \notin S'_i$. How could I have been added to S_i?

Case 1. I was added initially. In this case, $i = 0$, $I = [S' \to \bullet S, 0]$. Obviously, $I \in S'_i$ as well.

Case 2. I was added by E_S being applied to some item $I' \in S_{i-1}$. E_S is the same as E'_S and $S_{i-1} = S'_{i-1}$, so I must also be in S'_i.

Case 3. I was added by E_P. Say $I = [A \to \bullet\alpha, i]$. Then there must exist $I' = [B \to \ldots \bullet A \ldots, j] \in S_i$. If I' is also in S'_i, then $I \in S'_i$ because E'_P adds at least those items that E_P adds.

Case 4. I was added by E_C, operating on some $I' = [A \to \alpha\bullet, j] \in S_i$. Say $I = [B \to \ldots A \bullet \ldots, k]$. First, assume $j < i$. If $I' \in S'_i$, then $I \in S'_i$ because E_C and E'_C are the same, and would be referring back to Earley set j where $S_j = S'_j$. Now assume that $j = i$. By Lemma 5.1, $\alpha \Rightarrow^* \epsilon$ and thus $A \Rightarrow^* \epsilon$. There must therefore also be an item $I'' = [B \to \ldots \bullet A \ldots, k] \in S_i$. If $I'' \in S'_i$, then $I \in S'_i$, added by E'_P.

Therefore $S_i \subseteq S'_i$ by contradiction, since no I can be chosen such that $I \in S_i$ and $I \notin S'_i$. □

LEMMA 5.3. *If $S_0 = S'_0$, $S_1 = S'_1, \ldots, S_{i-1} = S'_{i-1}$, then $S_i \supseteq S'_i$.*

Proof. We take the same tack as before: posit the existence of an Earley item $I \in S'_i$ but $I \notin S_i$. How could I have been added to S'_i?

Case 1. I was added initially. This is the same situation as Case 1 of Lemma 5.2.

Case 2. I was added by E'_S. For this case, the proof is directly analogous to that given in Lemma 5.2, Case 2, and we therefore omit it.

Case 3. I was added by E'_P. If $I = [A \to \bullet\alpha, i]$, then this case is analogous to Case 3 of Lemma 5.2 and we omit it. Otherwise, $I = [A \to \ldots B \bullet \ldots, j]$. By definition of E'_P, $B \Rightarrow^* \epsilon$ and there is some $I' = [A \to \ldots \bullet B \ldots, j]$ in S'_i. If $I' \in S_i$, then $I \in S_i$ because, if it were not, Earley's algorithm would have failed to recognize that $B \Rightarrow^* \epsilon$.[4]

Case 4. I was added by E'_C. This case is analogous to Lemma 5.2, Case 4, and the proof is omitted.

As before, no I can be chosen such that $I \in S'_i$ and $I \notin S_i$, proving that $S_i \supseteq S'_i$ by contradiction. □

We can now state the main theorem.

THEOREM 5.1. $S_i = S'_i$, $0 \leq i \leq n$.

Proof. The proof follows from the previous two lemmas by induction. The basis for the induction is $S_0 = S'_0$. Assuming $S_k = S'_k$, $0 \leq k \leq i-1$, then $S_i = S'_i$ from Lemmas 5.2 and 5.3. □

Our solution thus produces the same item sets as Earley's algorithm.

6. THE BIRTH OF A NEW AUTOMATON

The Earley items added by our modified PREDICTOR can be precomputed. Moreover, this precomputation can take place in such a way as to produce a new variant of LR(0) deterministic finite automata (DFA) which is very well suited to Earley parsing.

In our description of Earley parsing up to now, we have not mentioned the use of automata. However, the idea to use efficient deterministic automata as the

[4] Earley's algorithm recognizes that B derives the empty string with a series of PREDICTOR and COMPLETER steps [16].

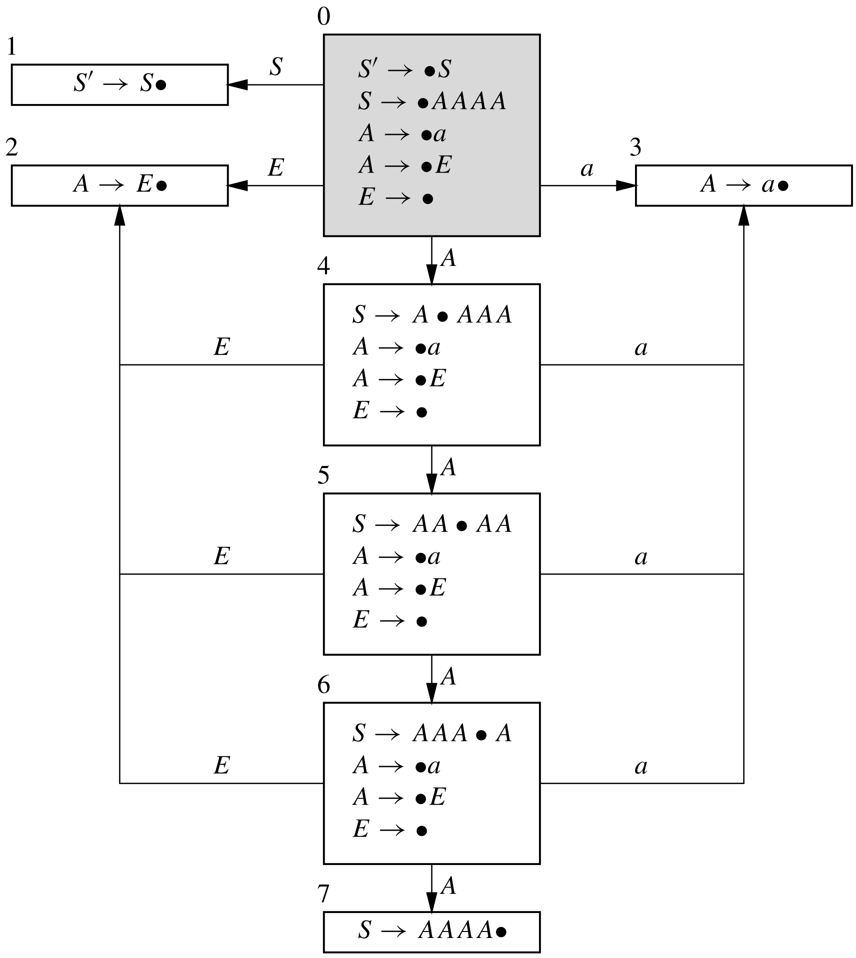

FIGURE 4. LR(0) automaton for the grammar in Figure 2. Shading indicates the start state.

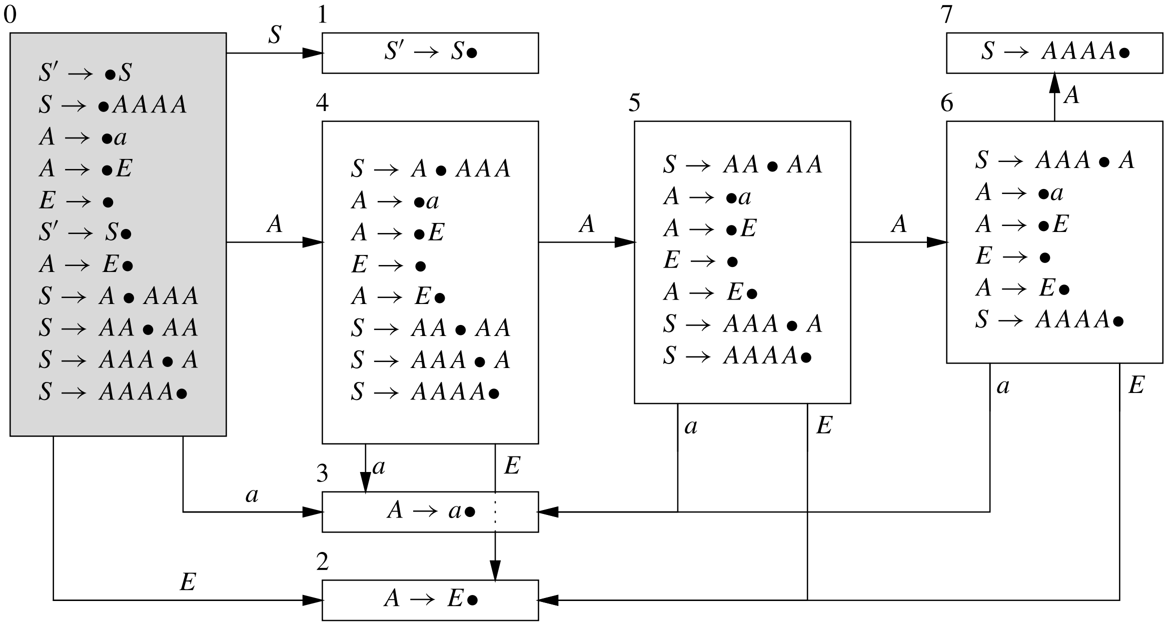

FIGURE 5. LR(0) ϵ-DFA.

basis for general parsing algorithms dates back to the early 1970s [17]. Traditional types of deterministic parsing automata (e.g. LR and LALR) have been applied with great success in Earley parsers [18] and Earley-like parsers [19, 20, 21]. Conceptually, this has the effect of precomputing groups of Earley items which must appear together in an Earley set, thus reducing the amount of work the Earley algorithm must perform at parse time. (We note that the parsing algorithm described in [16] is related to an Earley parser and employs precomputation as well.)

Figure 4 shows the LR(0) automaton for the grammar in Figure 2. (The construction of LR(0) automata is treated in most compiler texts [7] and is not especially pertinent to this discussion.) Each LR(0) automaton state consists of one or more LR(0) items, each of which is exactly the same as an Earley item less its parent pointer.

Assume that an Earley parser uses an LR(0) automaton. We discuss this in detail later, but for now it is sufficient to think of this as precomputing groups of items that appear together, as described above. We observe that some LR(0) states must always appear together in an Earley set. This idea is captured in the theorem below. The function $GOTO(L, A)$ returns the LR(0) state reached if a transition on A is made from LR(0) state L. Given an LR(0) item $l = [A \to \alpha \bullet \beta]$ and an Earley set S_i, $l \sqsubset S_i$ iff the Earley item $[A \to \alpha \bullet \beta, j] \in S_i$. Where L is a LR(0) state, we write $L \sqsubset S_i$ to mean $l \sqsubset S_i$ for all $l \in L$.

Lemma 6.1. *If $[B \to \bullet\alpha, i] \in S'_i$ and $\alpha \Rightarrow^* \epsilon$, then $[B \to \alpha\bullet, i]$ will be added to S'_i while performing the modified* Predictor *step.*

Proof. We look at the length of α.

Case 1. $|\alpha| = 0$. True, because $\alpha = \epsilon$.

Case 2. $|\alpha| > 0$. Let $m = |\alpha|$. Then $\alpha = X_1X_2 \ldots X_m$. Because $\alpha \Rightarrow^* \epsilon$, none of the constituent symbols of α can be terminals. Furthermore, $X_l \Rightarrow^* \epsilon, 1 \leq l \leq m$. When processing $[B \to \bullet X_1X_2 \ldots X_m, i]$, E'_{P} would add $[B \to X_1 \bullet X_2 \ldots X_m, i]$, whose later processing by E'_{P} would add $[B \to X_1X_2 \bullet \ldots X_m, i]$, and so on until $[B \to X_1X_2 \ldots X_m\bullet, i]$—also known as $[B \to \alpha\bullet, i]$—had been added to S'_i. □

Theorem 6.1. *If an LR(0) item $l = [A \to \bullet\alpha]$ is contained in LR(0) state L, $L \sqsubset S_i$ and $\alpha \Rightarrow^* \epsilon$, then $GOTO(L, A) \sqsubset S_i$.*

Proof. As part of an Earley item, l must have the parent pointer i because the dot is at the beginning of the item. By Lemma 6.1, the Earley item $I = [A \to \alpha\bullet, i]$ will be added to S_i. As a result of Completer processing of I (whose parent pointer is i, making Completer look 'back' at the current set S_i), transitions will be attempted on A for every LR(0) state in S_i. Thus $GOTO(L, A) \sqsubset S_i$. □

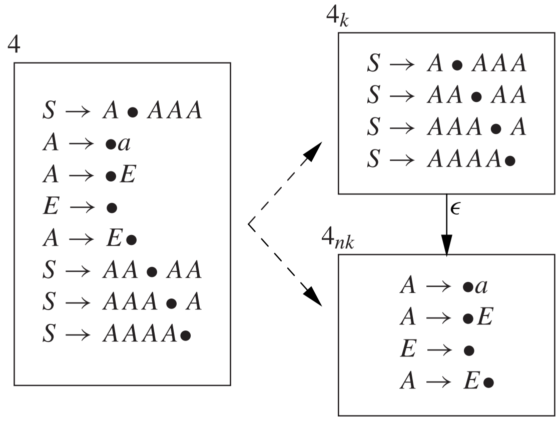

FIGURE 6. Splitting an LR(0) ϵ-DFA state into ϵ-kernel and ϵ-non-kernel states, joined by an ϵ edge.

We can treat Theorem 6.1 as the basis of a closure algorithm, combining LR(0) states that always appear together, iterating until no more state mergers are possible. This results in a new type of parsing automaton, which we call the 'LR(0) ϵ-DFA'. For the LR(0) automaton in Figure 4, the LR(0) ϵ-DFA states would be

$$\begin{array}{ll} \{0, 1, 2, 4, 5, 6, 7\} & \{1\} \\ \{4, 2, 5, 6, 7\} & \{2\} \\ \{5, 2, 6, 7\} & \{3\} \\ \{6, 2, 7\} & \{7\} \end{array}$$

The LR(0) ϵ-DFA is drawn in Figure 5.

Can the original problem with empty rules recur when using the ϵ-DFA in an Earley parser? Happily, it cannot. Recall that the problem with which we began this paper was the addition of an Earley item $[B \to \ldots \bullet A \ldots, k]$ to S_i *after* Completer processed $[A \to \bullet, i]$: the dot never got moved over the A even though $A \Rightarrow^* \epsilon$. With the ϵ-DFA, say that state l contains the troublesome item $[B \to \ldots \bullet A \ldots]$. All items $[A \to \bullet\alpha]$ must be in l too. If A is nullable, then there has to be some value of α such that $A \Rightarrow \alpha \Rightarrow^* \epsilon$ and the dot must be moved over the A in state l by Theorem 6.1.

7. PRACTICAL USE OF THE ϵ-DFA

We have not elaborated on the use of finite automata in Earley parsers because of a vexing implementation issue: the parser must keep track of which parent pointers and LR(0) items belong together. This leads to complex, inelegant implementations, such as an Earley item being represented by a tuple containing lists inside lists [18]. We have previously shown how to solve this problem by splitting the states of an LR(0) DFA and using a slightly *non*-deterministic LR(0) automaton instead [9]. This exploits the fact that an Earley parser is effectively simulating non-determinism.

This state-splitting idea may be extended to the ϵ-DFA. Each state s in the ϵ-DFA can be divided into (at most) two new states.

1. s_{nk}, the ϵ-non-kernel state. This contains all LR(0) items in s with the dot at the beginning of the right-hand side and any LR(0) items derived from them via nullability. One exception is that the start LR(0) item $[S' \to \bullet S]$ cannot be in an ϵ-non-kernel state.

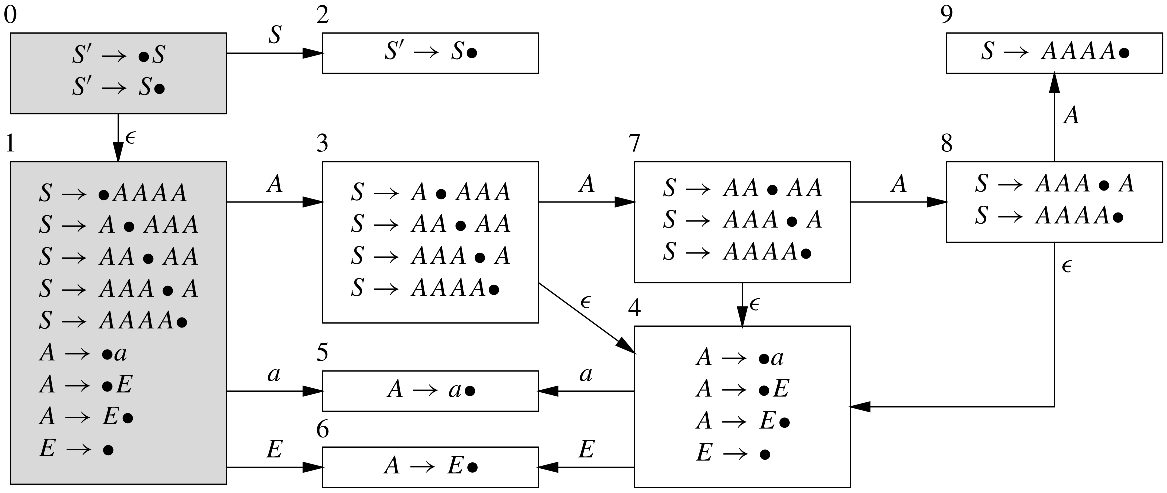

FIGURE 7. Split LR(0) ϵ-DFA.

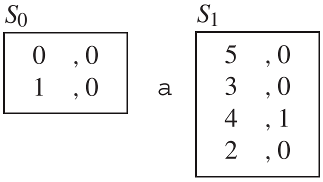

FIGURE 8. An Earley parser accepts the input a, using the split ϵ-DFA of Figure 7.

2. s_k, the ϵ-kernel state. All LR(0) items not in s_{nk} are contained in s_k. It is possible to have an ϵ-kernel state without a corresponding ϵ-non-kernel state, but not *vice versa*.

Figure 6 shows how state 4 of the ϵ-DFA in Figure 5 would be split. Clearly, the same information is being represented. Outgoing edges are retained accordingly with the LR(0) items in the new states. In Figure 6, state 4_{nk} would have a transition on a because it contains the item $[A \to \bullet a]$; state 4_k would not have such a transition, even though the original state 4 did. Incoming edges go to the ϵ-kernel state; the original edge coming into state 4 would now go to state 4_k. Having a transition on ϵ between ϵ-kernel and ϵ-non-kernel states gives a slightly non-deterministic automaton, which we call the split ϵ-DFA (Figure 7). Since an Earley parser simulates non-determinism, the same actions are being performed in the split ϵ-DFA as were occurring in the original DFA.

The advantage to splitting the states in this way is that maintaining parent pointers with the correct LR(0) items is now easy. All items in an ϵ-non-kernel state have the same parent pointer, as do all items in an ϵ-kernel state. In the former case, this is because the dot is at the beginning of each LR(0) item in the state (or derived via nullability from an item that did have the dot at the beginning); in the latter case, it is because an ϵ-kernel state is arrived at from a state which possessed this parent pointer property. Applying this result, it is now possible to represent an Earley item simply as a split ϵ-DFA state number and a parent pointer. This pair represents one or more of the original Earley items. Figure 8 shows the Earley sets from Figure 3 recoded using the split ϵ-DFA.

```
foreach (state, parent) in $S_i$:
    k ← goto(state, $x_{i+1}$)
    if k $\neq$ Λ:
        add (k, parent) to $S_{i+1}$
        nk ← goto(k, $\epsilon$)
        if nk $\neq$ Λ:
            add (nk, i+1) to $S_{i+1}$

    if parent = i:
        continue

    foreach $A \to \alpha$ in completed(state):
        foreach (pstate, pparent) in $S_{parent}$:
            k ← goto(pstate, $A$)
            if k $\neq$ Λ:
                add (k, pparent) to $S_i$
                nk ← goto(k, $\epsilon$)
                if nk $\neq$ Λ:
                    add (nk, i) to $S_i$
```

FIGURE 9. Pseudocode for processing Earley set S_i using a split ϵ-DFA.

How would an Earley parser use the split ϵ-DFA? Some pseudocode is given in Figure 9. This code assumes that Earley items in S_i and S_{i+1} are implemented as worklists and that the items themselves are (state, parent) pairs. The goto function returns a state transition made in the split ϵ-DFA, given a state and grammar symbol (or ϵ). The absence of a transition is denoted by the value Λ. There is at most a single transition in every case, so goto may be implemented as a simple table lookup. The completed function supplies a list of grammar rules completed within a given state. Recall that an Earley item is only added to an Earley set if it is not already present. Both goto and completed are computed in advance.

The work for each Earley item now divides into two parts. First, we add Earley items to S_{i+1} resulting from moving the dot over a terminal symbol; this is the SCANNER. Second, we ascertain which grammar rules are now complete and attempt to move the dot over the corresponding non-terminal in parent Earley sets; this is the COMPLETER. A special case exists here, because it is now unnecessary to complete items which begin and end in the current set, S_i. In our split ϵ-DFA, this is all precomputed, as is all the work of PREDICTOR.

While use of our split ϵ-DFA makes Earley parsing more efficient in practice, as we will show in Section 9, we have not changed the underlying time complexity of Earley's algorithm. In the worst case, each split ϵ-DFA state would contain a single item, effectively reducing it to Earley's original algorithm. Having said this, we are not aware of any practical example where this occurs.

8. RECONSTRUCTING DERIVATIONS

To this point, we have described efficient recognition, but practical use of Earley's algorithm demands that we be able to reconstruct derivations of the input. Finding a good method to do this is a harder problem than it seems.

At one level, the argument can be made that our new Earley items—consisting of a split ϵ-DFA state and a parent pointer—contain the same information as a standard Earley parser, just in compressed form. Therefore, by retaining information about the LR(0) items in each ϵ-DFA state, we could reconstruct derivations as a standard Earley parser would. Of course, this method would be unsatisfying in the extreme. It seems silly to sift through the contents of a state at parse time and to retain the internal information about a state in the first place. We would like a method which operates at the granularity of states.

Compounding the problem is that a lot of activity can occur within a single ϵ-DFA state due to ϵ-rules. Consider our running example, the grammar whose split LR(0) ϵ-DFA is shown in Figure 7: given the legal input ϵ, ten derivation steps would have to be reconstructed from one Earley set containing only two items!

Our solution is in two parts. First, we follow Earley [1, 2] and add links between Earley items to track information about an item's pedigree. As the contents of our Earley items are different from that of a standard Earley parser, it is worth elaborating on this. Our Earley items can have two types of links.

1. Predecessor links. If an item (s, p) is added as a result of a state machine transition from s_p to s, where s_p is part of the item (s_p, p_p), then we add a predecessor link from (s, p) to (s_p, p_p).
2. Causal links. Why did a transition from s_p to s occur? There can be three reasons.
   (a) Transition on a terminal symbol. We record the causal link for (s, p) as being absent (Λ).
   (b) ϵ-transition in the state machine. We need not store a causal link *or* a predecessor link for (s, p) in this case.
   (c) Transition on a non-terminal symbol. In this case, (s, p) was added as the result of a rule completion in some state s_c; we add a causal link from (s, p) to (s_c, p_c).

Both types of links are easy to compute at parse time.

TABLE 1. Rule increase in practical grammars due to NNF conversion.

	Rules	ϵ-rules	NNF rules
Ada	459	52	701
Algol 60	169	7	191
C++	643	15	794
C	214	0	214
Delphi	529	34	651
Intercal	87	1	89
Java 1.1	269	0	269
Modula-2	245	33	326
Pascal	178	10	219
Pilot	68	3	125

Second, we do not create the split ϵ-DFA using the original grammar G, but an equivalent grammar G_ϵ which explicitly encodes missing information that is crucial to reconstructing derivations. As G_ϵ deals with non-existence, as it were, we call it the nihilist normal form (NNF) grammar.

The NNF grammar G_ϵ is straightforward to produce. For every non-terminal A which is nullable, we create a new non-terminal symbol A_ϵ in G_ϵ. Then, for each rule in G in which A appears on the right-hand side, we create rules in G_ϵ that account for the nullability of A. For example, a rule $B \to \alpha A \beta$ in G would beget the rules $B \to \alpha A \beta$ and $B \to \alpha A_\epsilon \beta$ in G_ϵ. In the event of multiple nullable non-terminals in a rule's right-hand side, all possible combinations must be enumerated. Finally, if a rule's right-hand side is empty or if all of its right-hand side is populated by 'ϵ-non-terminals', then we replace its left-hand side by the corresponding ϵ-non-terminal. For instance, $A \to \epsilon$ would become $A_\epsilon \to \epsilon$, and $A \to B_\epsilon C_\epsilon D_\epsilon$ would become $A_\epsilon \to B_\epsilon C_\epsilon D_\epsilon$. The new grammar for our running example is:

$$\begin{array}{lcl@{\qquad}lcl}
S' & \to & S & S & \to & A_\epsilon AAA \\
S'_\epsilon & \to & S_\epsilon & S & \to & A_\epsilon AAA_\epsilon \\
S & \to & AAAA & S & \to & A_\epsilon AA_\epsilon A \\
S & \to & AAAA_\epsilon & S & \to & A_\epsilon AA_\epsilon A_\epsilon \\
S & \to & AAA_\epsilon A & S & \to & A_\epsilon A_\epsilon AA \\
S & \to & AAA_\epsilon A_\epsilon & S & \to & A_\epsilon A_\epsilon AA_\epsilon \\
S & \to & AA_\epsilon AA & S & \to & A_\epsilon A_\epsilon A_\epsilon A \\
S & \to & AA_\epsilon AA_\epsilon & S_\epsilon & \to & A_\epsilon A_\epsilon A_\epsilon A_\epsilon \\
S & \to & AA_\epsilon A_\epsilon A & A & \to & a \\
S & \to & AA_\epsilon A_\epsilon A_\epsilon & A_\epsilon & \to & E_\epsilon \\
 & & & E_\epsilon & \to & \epsilon
\end{array}$$

The size of an NNF grammar is, as might be expected, dependent on the use of ϵ-rules in the original grammar. We took ten programming language grammars from two repositories[5] to see what effect this would have in practice. The results are shown in Table 1. The conversion to an NNF grammar did not even double the number of grammar rules; most increases were quite modest. The grammar

[5] The comp.compilers archive (http://www.iecc.com) and the Retrocomputing Museum (http://www.tuxedo.org/~esr/retro).

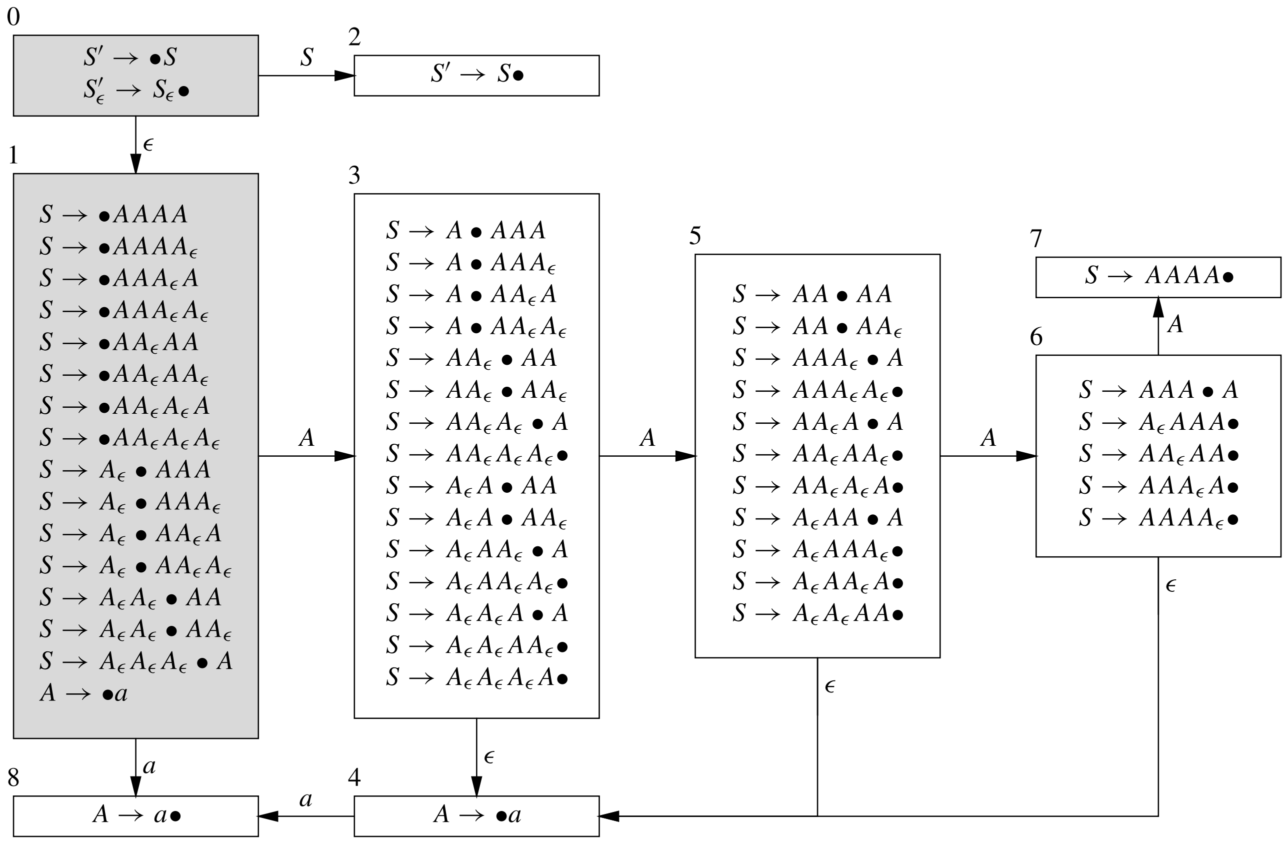
0
$S' \to \bullet S$
$S'_\epsilon \to S_\epsilon \bullet$
S
2
$S' \to S\bullet$
ϵ
1
$S \to \bullet AAAA$
$S \to \bullet AAAA_\epsilon$
$S \to \bullet AAA_\epsilon A$
$S \to \bullet AAA_\epsilon A_\epsilon$
$S \to \bullet AA_\epsilon AA$
$S \to \bullet AA_\epsilon AA_\epsilon$
$S \to \bullet AA_\epsilon A_\epsilon A$
$S \to \bullet AA_\epsilon A_\epsilon A_\epsilon$
$S \to A_\epsilon \bullet AAA$
$S \to A_\epsilon \bullet AAA_\epsilon$
$S \to A_\epsilon \bullet AA_\epsilon A$
$S \to A_\epsilon \bullet AA_\epsilon A_\epsilon$
$S \to A_\epsilon A_\epsilon \bullet AA$
$S \to A_\epsilon A_\epsilon \bullet AA_\epsilon$
$S \to A_\epsilon A_\epsilon A_\epsilon \bullet A$
$A \to \bullet a$
A
3
$S \to A \bullet AAA$
$S \to A \bullet AAA_\epsilon$
$S \to A \bullet AA_\epsilon A$
$S \to A \bullet AA_\epsilon A_\epsilon$
$S \to AA_\epsilon \bullet AA$
$S \to AA_\epsilon \bullet AA_\epsilon$
$S \to AA_\epsilon A_\epsilon \bullet A$
$S \to AA_\epsilon A_\epsilon A_\epsilon \bullet$
$S \to A_\epsilon A \bullet AA$
$S \to A_\epsilon A \bullet AA_\epsilon$
$S \to A_\epsilon AA_\epsilon \bullet A$
$S \to A_\epsilon AA_\epsilon A_\epsilon \bullet$
$S \to A_\epsilon A_\epsilon A \bullet A$
$S \to A_\epsilon A_\epsilon AA_\epsilon \bullet$
$S \to A_\epsilon A_\epsilon A_\epsilon A\bullet$
A
5
$S \to AA \bullet AA$
$S \to AA \bullet AA_\epsilon$
$S \to AAA_\epsilon \bullet A$
$S \to AAA_\epsilon A_\epsilon \bullet$
$S \to AA_\epsilon A \bullet A$
$S \to AA_\epsilon AA_\epsilon \bullet$
$S \to AA_\epsilon A_\epsilon A\bullet$
$S \to A_\epsilon AA \bullet A$
$S \to A_\epsilon AAA_\epsilon \bullet$
$S \to A_\epsilon AA_\epsilon A\bullet$
$S \to A_\epsilon A_\epsilon AA\bullet$
A
7
$S \to AAAA\bullet$
A
6
$S \to AAA \bullet A$
$S \to A_\epsilon AAA\bullet$
$S \to AA_\epsilon AA\bullet$
$S \to AAA_\epsilon A\bullet$
$S \to AAAA_\epsilon \bullet$
ϵ
ϵ
ϵ
a
8
$A \to a\bullet$
a
4
$A \to \bullet a$

FIGURE 10. The new split LR(0) ϵ-DFA.

conversion and state machine construction would usually occur at compiler build time, where execution speed is not such a critical issue. However, as our results in the next section show, even when we convert the grammar and construct the state machine lazily at run-time, our method is still substantially faster (over 40% faster) then a standard Earley parser.

When constructing the split ϵ-DFA from G_ϵ, ϵ-non-terminals such as A_ϵ in the right-hand side of a grammar rule act as ϵ symbols in that the dot is always moved over them immediately. Both S' and S'_ϵ, if present, are treated as start symbols and it is important to note that two symbols A and A_ϵ are treated as distinct. Figure 10 shows the new split ϵ-DFA. Some states, and LR(0) items within states, that were present in Figure 7 are now absent. These missing states and items were extraneous and will be inferred instead.

To be explicit about how the parser is modified to attach predecessor and causal links to the items in an Earley set, new pseudocode for processing an Earley set is shown in Figure 11. The pseudocode assumes that each item has an associated set of links, where that set is organized as a set of (predecessor link, causal link) pairs; the notation `&x` is used to denote a link to the item `x`.

```
foreach item in $S_i$:
    (state, parent) ← item
    k ← goto(state, $x_{i+1}$)
    if k ≠ Λ:
        add (k, parent) to $S_{i+1}$
        add link (&item, Λ) to ↩
          (k, parent) in $S_{i+1}$
        nk ← goto(k, $\epsilon$)
        if nk ≠ Λ:
            add (nk, i+1) to $S_{i+1}$

    if parent = i:
        continue

    foreach $A \to \alpha$ in completed(state):
        foreach pitem in $S_{parent}$:
            (pstate, pparent) ← pitem
            k ← goto(pstate, $A$)
            if k ≠ Λ:
                add (k, pparent) to $S_i$
                add link (&pitem, &item) to ↩
                  (k, pparent) in $S_i$
                nk ← goto(k, $\epsilon$)
                if nk ≠ Λ:
                    add (nk, i) to $S_i$
```

FIGURE 11. Pseudocode for processing Earley set S_i with creation of predecessor and causal links ('↩' is a line continuation symbol).

Pseudocode for constructing a rightmost derivation is given in Figure 12. With an ambiguous grammar, there may be multiple derivations for a given input; the shaded lines mark places where a choice between several alternatives may need to be made. These choices may be made using heuristics, user-defined routines, disambiguating rules [22] or more elaborate means [23].

Assuming some disambiguation mechanism, the `causal` function returns the item pointed to by a causal link and the `predecessor` function returns the predecessor of an item,

```
def rparse($A$, item):
    (state, parent) ← item
    choose $A \to X_1 X_2 \dots X_n$ in completed(state)
    output $A \to X_1 X_2 \dots X_n$
    for sym in $X_n, X_{n-1}, \dots, X_1$:
        if sym is $\epsilon$-non-terminal:
            derive-$\epsilon$(sym)
        else if sym is terminal:
            item ← predecessor(item, $\Lambda$)
        else:
            item$_{why}$ ← causal(item)
            rparse(sym, item$_{why}$)
            item ← predecessor(item, item$_{why}$)
```

FIGURE 12. Pseudocode for constructing a rightmost derivation.

```
rparse($S'$, (2,0))          $S' \to S$
  rparse($S$, (5,0))         $S \to AAAA$    †
    derive-$\epsilon$($A_\epsilon$)          $A \to E$
                             $E \to \epsilon$
    rparse($A$, (8,1))       $A \to a$
    derive-$\epsilon$($A_\epsilon$)          $A \to E$
                             $E \to \epsilon$
    rparse($A$, (8,0))       $A \to a$
```

FIGURE 13. A trace of rparse; the alternative $S \to AA_\epsilon AA_\epsilon$ has been chosen at (†).

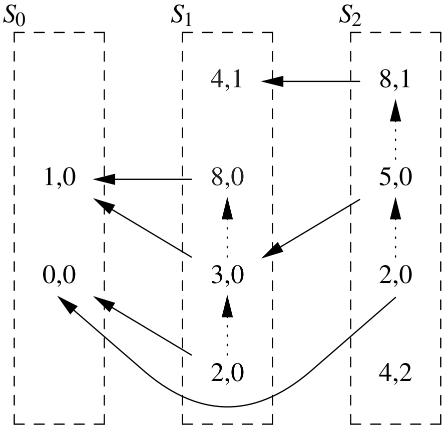

FIGURE 14. Earley sets and items for the input aa. Predecessor links are represented by solid arrows and causal links by dotted arrows. Some Earley items within a set have been reordered for aesthetic reasons.

TABLE 2. SPARK timings.

	Time (s)
Earley	2132.81
PEP (lazy)	1497.37
PEP (precomputed)	1050.77

given its associated causal item (possibly Λ). How rparse is initially invoked depends on the input: rparse(S'_ϵ,I) if the input is ϵ and rparse(S',I) otherwise. The second argument, I, is the Earley item in the final Earley set containing the LR(0) item $[S'_\epsilon \to S_\epsilon \bullet]$ or $[S' \to S\bullet]$, as appropriate. We make the tacit assumption that I identifies a single (Earley item, Earley set) combination as might be the case with an implementation using pointers.

The purpose of derive-ϵ is to output the rightmost derivation of ϵ from a given ϵ-non-terminal. If there is only one way to derive the empty string, this may be precomputed so that derive-ϵ need only output the derivation sequence.

Finally, rules that are output in any fashion should be mapped back to rules in the original grammar G, a trivial operation. Figure 13 traces the operation of rparse on the Earley items and links in Figure 14.

Rather than use NNF, another approach would be to systematically remove ϵ-rules from the original grammar, then apply Earley's algorithm to the resulting ϵ-free grammar. For our running example, the corresponding ϵ-free grammar would be:

$$\begin{array}{lcl} S' & \to & S \\ S' & \to & \epsilon \\ S & \to & AAAA \\ S & \to & AAA \\ S & \to & AA \\ S & \to & A \\ A & \to & a \end{array}$$

While this is an equivalent grammar to our NNF one, in the formal sense, there is a substantial amount of information lost. For example, when the rule $S \to AAA$ is recognized using the ϵ-free grammar above, it is not clear that there were *four* ways of arriving at this conclusion in the original grammar, that an ambiguity in the original grammar has been encountered and when any semantic actions associated with the original grammar rules $A \to E$ and $E \to \epsilon$ should be invoked. Clearly, having an equivalent grammar does not imply honoring the structure of the original grammar as with our approach. This is a matter of practical import, because the users of a parser will expect to have the parser behave as though it were using the grammar the user specified.

One area of future work is to try and find better ways to reconstruct derivations.

9. EXPERIMENTAL RESULTS

We have two independently-written implementations of our recognition and reconstruction algorithms in the last two sections, which we will refer to as 'PEP'. The timings below were taken on a 700 MHz Pentium III with 256 MB RAM running Red Hat Linux 7.1, and represent combined user and system times.

One implementation of PEP is in the SPARK toolkit. We scanned and parsed 735 files of Python source code, 786,793 tokens in total; the results are shown in Table 2. There is the standard Earley parser, of course, and two versions of PEP: one which precomputes the split LR(0) ϵ-DFA and another which lazily constructs the state machine (similar to the approach taken in [24] for GLR parsing). All timings reported in this section include both the time to recognize the input and the time to construct a rightmost derivation. The precomputed version of PEP runs twice as fast as the standard Earley algorithm, the lazy version about 42% faster.

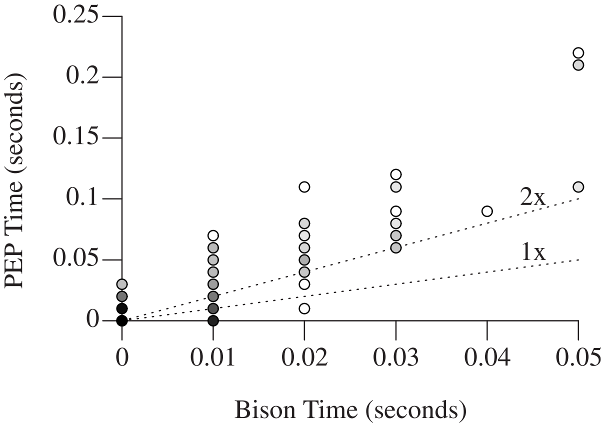

FIGURE 15. PEP versus Bison timings on Java source files from JDK 1.2.2. Darker circles indicate many points plotted at that spot.

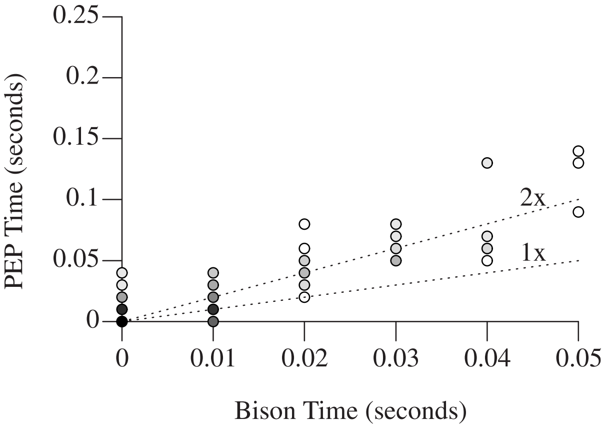

FIGURE 16. PEP versus Bison timings on a faster machine with more memory.

The other implementation of PEP is written in C, allowing a comparison with LALR(1) parsers generated by Bison, a Yacc-compatible parser generator. We used the same (Flex-generated) lexical analyzer for both and compiled everything with `gcc -O`. Figure 15 shows how the parsers compared parsing 3234 Java source files from JDK 1.2.2, plotting the sum of user and system times for each parser. Due to clock granularity, data points only appear at intervals, and many points fell at the same spot; we have used greyscaling to indicate point density, where a darker color means more points. Overall, 97% of PEP times were 0.02 s or less. Looking at cumulative time over the entire test suite, PEP was only 2.1 times slower than the much more specialized LALR(1) parser of Bison.

To look at how PEP might be affected by the computing environment, we repeated the Java test run, this time using a 1.4 GHz Pentium IV XEON with 1 GB RAM running Red Hat Linux 7.1. The results, in Figure 16, show that individual PEP times drop noticeably: the worst case measured was 0.08 s less than before. The cumulative time remained roughly the same, with PEP only 1.9 times slower than Bison. Given the generality of Earley parsing compared to LALR(1) parsing, and considering that even PEP's worst time would not be noticeable by a user, this is an excellent result.

10. CONCLUSION

Implementations of Earley's parsing algorithm can easily handle ϵ-rules using the simple modification to PREDICTOR outlined here. Precomputation leads to a new variant of LR(0) state machine tailored for use in Earley parsers based on finite automata. Timings show that our new algorithm is twice as fast as a standard Earley parser and fares well compared to more specialized parsing algorithms.

ACKNOWLEDGEMENTS

This work was supported in part by a grant from the National Science and Engineering Research Council of Canada. This paper was greatly improved thanks to comments from the referees and Angelo Borsotti.

REFERENCES

[1] Earley, J. (1968) *An Efficient Context-Free Parsing Algorithm*. PhD Thesis, Carnegie-Mellon University.

[2] Earley, J. (1970) An efficient context-free parsing algorithm. *Commun. ACM*, **13**, 94–102.

[3] Johnson, S. C. (1978) Yacc: yet another compiler–compiler. *Unix Programmer's Manual* (7th edn), Vol. 2B. Bell Laboratories, Murray Hill, NJ.

[4] Willink, E. D. (2000) Resolution of parsing difficulties. *Meta-Compilation for C++*. PhD Thesis, University of Surrey, Section F.2, pp. 336–350.

[5] van den Brand, M., Sellink, A. and Verhoef, C. (1998) Current parsing techniques in software renovation considered harmful. In *Proc. 6th Int. Workshop on Program Comprehension*, Ischia, Italy, June 24–26, pp. 108–117. IEEE Computer Society Press, Los Alamitos, CA.

[6] Glanville, R. S. and Graham, S. L. (1978) A new method for compiler code generation. In *Proc. Fifth Ann. ACM Symp. on Principles of Programming Languages*, Tucson, AZ, January 23–25, pp. 231–240. ACM Press, New York.

[7] Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

[8] Bouckaert, M., Pirotte, A. and Snelling, M. (1975) Efficient parsing algorithms for general context-free parsers. *Inform. Sci.*, **8**, 1–26.

[9] Aycock, J. and Horspool, N. (2001) Directly-executable Earley parsing. In *CC 2001—10th International Conference on Compiler Construction*, Genova, Italy, April 2–6. *Lecture Notes in Computer Science*, **2027**, 229–243. Springer, Berlin.

[10] Grune, D. and Jacobs, C. J. H. (1990) *Parsing Techniques: A Practical Guide*. Ellis Horwood, New York.

[11] Aho, A. V. and Ullman, J. D. (1972) *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, NJ.

[12] Schröer, F. W. (2000) *The ACCENT Compiler Compiler, Introduction and Reference*. GMD Report 101, German National Research Center for Information Technology.

[13] Aycock, J. (1998) Compiling little languages in Python. In *Proc. 7th Int. Python Conf.*, Houston, TX, November 10–13, pp. 69–77. Foretec Seminars, Reston, VA.

[14] Appel, A. W. (1998) *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, UK.

[15] Fischer, C. N. and LeBlanc Jr, R. J. (1988) *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA.

[16] Graham, S. L., Harrison, M. A. and Ruzzo, W. L. (1980) An improved context-free recognizer. *ACM Trans. Program. Languages Syst.*, **2**, 415–462.

[17] Lang, B. (1974) Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages, and Programming*, Saarbrücken, July 29–August 2. *Lecture Notes in Computer Science*, **14**, 255–269. Springer, Berlin.

[18] McLean, P. and Horspool, R. N. (1996) A faster Earley parser. In *Proc. 6th Int. Conf. on Compiler Construction*, Linköping, Sweden, April 24–26. *Lecture Notes in Computer Science*, **1060**, 281–293. Springer, Berlin.

[19] Billot, S. and Lang, B. (1989) The structure of shared forests in ambiguous parsing. In *Proc. 27th Ann. Meeting of the Association for Computational Linguistics*, Vancouver, Canada, June 26–29, pp. 143–151. Association for Computational Linguistics, Morristown, NJ.

[20] Vilares Ferro, M. and Dion, B. A. (1994) Efficient incremental parsing for context-free languages. In *Proc. 5th IEEE Int. Conf. on Computer Languages*, Toulouse, France, May 16–19, pp. 241–252. IEEE Computer Society Press, Los Alamitos, CA.

[21] Alonso, M. A., Cabrero, D. and Vilares, M. (1997) Construction of efficient generalized LR parsers. In *Proc. 2nd Int. Workshop on Implementing Automata*, London, Canada, September 18–20. *Lecture Notes in Computer Science*, **1436**, 131–140. Springer, Berlin.

[22] Aho, A. V., Johnson, S. C. and Ullman, J. D. (1975) Deterministic parsing of ambiguous grammars. *Commun. ACM*, **18**, 441–452.

[23] Klint, P. and Visser, E. (1994) Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*, Milano, Italy, October 12–14, pp. 1–20. Technical Report 12694, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano.

[24] Heering, J., Klint, P. and Rekers, J. (1989) Incremental generation of parsers. In *Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation*, Portland, OR, June 21–23, pp. 179–191. ACM Press, New York.