A Faster Earley Parser

Philippe McLean & R. Nigel Horspool

Dept. of Computer Science, University of Victoria
Victoria, BC, Canada V8W 3P6
E-mail: pmclean@csc.uvic.ca, nigelh@csc.uvic.ca

Abstract. We present a parsing technique which is a hybrid of Earley's method and the LR(k) methods. The new method retains the ability of Earley's method to parse using arbitrary context-free grammars. However, by using precomputed LR(k) sets of items, we obtain much faster recognition speeds while also reducing memory requirements.

1 Introduction

The parsing method invented by Earley [2,4] is a highly practical parsing technique for general context-free grammars (CFGs). If n is the length of the input to be recognized, the parser requires time proportional to n^3 to recognize arbitrary context-free languages, n^2 for unambiguous languages, and n for a large class of languages.

The amount of processing performed while recognizing an input string is large compared to table-driven techniques such as the LR parser family, which includes the LR(0), SLR(1), LALR(1) and LR(1) methods. These LR methods, however, cannot accept arbitrary CFGs. They are limited to subsets of unambiguous grammars. In general, the LR parsing table constructed for an arbitrary CFG will contain conflicts. That is, one or more states will provide a choice of actions to perform for some inputs.

A parsing method due to Tomita [6,4] overcomes the limitations of the LR methods. It uses LR tables that may contain conflicts. Whenever the parser encounters a choice of parsing actions, it in effect clones new copies of itself to track each of the conflicting actions simultaneously. Some copies of the parser may subsequently reach a state where parsing cannot proceed (i.e. the input symbol is invalid for that state) and these copies of the parsers simply terminate execution. In practice, the Tomita parser simulates parallel execution of multiple copies of a LR parser, and it uses a DAG data structure to reduce the storage needed by all the parse stacks. A Tomita parser is particularly efficient when few conflicts are encountered in the LR states.

If all we need to do is recognize the input, a Tomita parser would likely be the method of choice. However, we will usually wish to execute semantic actions while precisely *one* of the parses is being performed. This is not so easy for a Tomita parser because many parses are being performed in parallel. One possible solution is for each copy of the LR parser to construct a parse tree. At the end of the input, we can traverse one of these parse trees to perform the desired semantic actions. We consider that the

This document was created with FrameMaker 4.0.4

computational work of building the parse trees negates the advantage of Tomita's method.

The Earley parser builds a data structure, a threaded sequence of states, which represents all possible parses of the input. After the input has been processed, it is straightforward to traverse the sequence of states to build a parse tree for one possible parse of the input, or to execute semantic actions for just the one parse.

We have developed a variation on Earley's method which, like Tomita's method, uses LR parse tables for efficiency, while retaining the advantage of permitting semantic actions to be easily associated with the grammar. The LR tables, in effect, capture precomputations of all the run-time actions performed by an Earley parser. Our parsing method, which we call LRE(*k*), uses information from the LR tables and therefore avoids recomputing this information at run-time. The name LRE(*k*) reflects the fact that our method can be viewed as a combination of LR(*k*) parsing with Earley parsing.

2 Terminology and Notation

2.1 Context-Free Grammars

A context free grammar G is a four-tuple $\langle V_T, V_N, P, Start \rangle$ where V_T is a set of terminal symbols, V_N is a set of nonterminal symbols, $V_N \cap V_T = \emptyset$, P is a set of productions, and $Start \in V_N$ is the start symbol or goal symbol of the grammar. The vocabulary $V = V_N \cup V_T$.

An augmented grammar G' is formed from G by adding a special goal rule

$$G' = \langle V_T \cup \{ \dashv \},\ V_N \cup \{ S' \},\ P \cup \{ Start' \rightarrow \vdash Start \dashv \},\ Start' \rangle.$$

where the tokens $\vdash$ and $\dashv$ are delimiters that represent the beginning and end of input.

Lower-case letters near the front of the alphabet (i.e. a, b, c ...) represent elements of V_T, upper-case letters near the front of the alphabet (i.e. A, B, C ...) represent elements of V_N, and upper-case letters near the end of the alphabet (i.e. X, Y, Z) represent elements of V. A superscript represents repetitions of a symbol, so that, for example, a^3 represents the string *aaa*. Greek letters α, β, ... represent sequences of zero or more vocabulary symbols.

2.2 LR(*k*) Recognizers

An *item* is a production which contains a marker, written as a dot, to indicate how much of the right-hand side (RHS) has been recognized. Associated with each item is a string of k symbols ($k \geq 0$). The string represents lookahead or right context for the production. For example, if k is 2, a possible item is [A → a b • B c, dd]. This item indicates that we have matched the first two symbols on the right-hand side of the rule A → a b B c. If the complete RHS is successfully matched, then the next two symbols in the input should be dd for this production to be valid in a parse of the input at this point.

We use S to denote the set of LR(k) sets of items for the augmented grammar G′. Each element of S corresponds to a state in the LR(k) recognizer for G′. The recognizer has an initial state

$I_{initial} = \{ [\ Start' \rightarrow \bullet \vdash Start \dashv, \dashv^k\ ] \} \in S$,

and it has an accept state

$I_{accept} = \{ [\ Start' \rightarrow \vdash Start \dashv \bullet, \dashv^k\ ] \} \in S$.

The transition function between the recognizer's states is

goto : $S \times V \rightarrow S \cup \{\varnothing\}$

The function *goto(I, x)* is defined as the set of all items $[\ A \rightarrow \alpha\ x \bullet \beta, t_1 \ldots t_k\ ]$ such that $[A \rightarrow \alpha \bullet x\ \beta, t_1 \ldots t_k\ ] \in I$. If the set *goto(I,x)* is an empty set, the transition is illegal. (I.e., the string $x\ t_1 \ldots t_k$ cannot follow the symbols that have been accepted so far in a syntactically valid input.)

The closure of an itemset I is defined as the least set J such that $I \subseteq J$, and $[\ A \rightarrow \alpha \bullet B\ \beta, t_1 \ldots t_k\ ] \in J$ implies that $\forall \eta\ (\eta \in first_k(\beta, t_1 \ldots t_k))$: $\{ [\ B \rightarrow \bullet\ \gamma, \eta] \mid B \rightarrow \gamma \in P \} \subseteq J$.

The function $first_k(\beta, \gamma) \equiv_{def} \{ prefix_k(\sigma) \mid \beta\gamma \Rightarrow^* \sigma,\ \sigma \in V_T^* \}$, where $prefix_k(\sigma)$ is the *k*-symbol prefix of σ.

The set of items for each state may be partitioned into kernel items and non-kernel items. The former are those items which are not added to a state by closure, while the latter (also called completion items) are those which are added to a state by closure.

3 Conventional Earley Recognizers

A conventional Earley recognizer has two inputs: a context-free grammar G and a token string $x_1\ x_2 \ldots x_n$, and determines if the string may be derived by G. For simplicity, lookahead will not be considered in this discussion *(k* = 0).

The recognizer constructs a sequence $E_1, E_2 \ldots, E_{n+1}$, of sets of tuples. Each tuple has the form <i, p> where *i* is an item $[\ A \rightarrow \alpha\ \bullet\ \beta\ ]$ and *p* is an integer referring to the parent Earley set E_p where the tuple containing the item with the marker at the beginning of the RHS was introduced. The *k*-th set is formed as a result of recognizing the first *k*-1 input tokens.

Tuples in a state may be partitioned into active and predicted tuples. Active tuples may be introduced in two ways: by a *SCANNER* operation, and by a *COMPLETER* operation. The SCANNER operation introduces tuples from the previous state where the marker appears before the current input token; the marker is advanced past that token in the new item. This is the process of matching terminal tokens in a production's RHS, and corresponds to a shift operation in an LR parser. The COMPLETER operation identifies each tuple where an item's marker is at the end of a RHS, and moves the marker past the LHS in items in the tuple's parent state. This operation identifies the derivation of a non-terminal, in the recognition of some RHS; an LR parser would perform a reduction in exactly this case.

The COMPLETER operation introduces new tuples for every item where the marker appears before a non-terminal. This operation begins the recognition of possible derivations for a non-terminal; it is the closure of a set of items. Closure is performed at parse time in a conventional Earley parser. However these closure items are implicit in the LR(*k*) recognizer.

Earley's doctoral dissertation [3] contains a proof of correctness for a conventional Earley recognizer, and an analysis of its algorithmic complexity. Parse trees may be enumerated for all derivations of the input string by examining the sets E_i, $1 \le i \le n+1$.

The conventional recognizer affords a simple implementation. However, observation of the parser's actions reveals that the parser spends much of its time introducing new items during the completion operation. Many prediction items may not be used during the parse. The computation of item-set closures, a grammar-dependent operation, is performed at parse time. It is natural to wonder whether the Earley items can be grouped in a manner that exploits pre-computed properties of the grammar. Our solution is to group items into sets in exactly the same way as in the states of a deterministic (and possibly inadequate) LR(*k*) finite-state automaton.

4 LRE – A Faster Earley Recognizer

The new parsing method is named LRE(*k*); this represents the hybrid nature of the algorithm as a composition of the LR(*k*) and Earley parsing methods.

In the following description, we use $x_1\ x_2\ ...\ x_n$ to represent the input to the recognizer. So that lookahead sets are properly defined, we assume that the input is terminated by *k* end-of-file delimiters. I.e., $x_{n+i} = \dashv$, for $1 \le i \le k$.

Our algorithm is based on a conventional Earley parser and its correct operation may be established by comparing its actions to an Earley parser's actions. A conventional Earley parser uses items of the form $[\ A \rightarrow \alpha \bullet \beta,\ t_1\ ...\ t_k,\ p\ ]$, where $A \rightarrow \alpha \bullet \beta$ is a marked production, $t_1\ ...\ t_k$ is the lookahead for the item, and *p* is a reference back to the state where recognition of the rule $A \rightarrow \alpha\ \beta$ commenced. Our algorithm takes advantage of the fact that the first two components of the Earley item represent an item in one or more states of the LR(*k*) recognizer. We therefore implement states in our LRE parser in terms of states in the LR(*k*) recognizer. The advantages of our representation are (1) we can use the LR(*k*) recognizer's tables to determine actions for the Earley parser, (2) the lookahead strings are not computed dynamically, and (3) the new representation can be implemented in a manner which uses much less storage.

A state in our LRE recognizer will be called an *Earley state*, and will be written as E_m. State E_m is reached after recognizing the token string $x_1\ x_2\ ...\ x_{m-1}$. The state E_m is represented by a set of tuples $\{\ \langle I_1, B_1\rangle, \langle I_2, B_2\rangle, ...\ \}$ where each $I_i \in S$ is the number of some state in the LR(*k*) recognizer and B_i is an organized collection of back-pointers to Earley states. In programming terms, each B_i could be implemented as an array of lists of LRE state numbers, where elements in the array are in one-to-one correspondence with items in LR(*k*) state I_i. In more formal terms, we can represent B_i as a list of list of integers $[\ [b_{i\,1\,1}, b_{i\,1\,2}, b_{i\,1\,3}\ ...], [b_{i\,2\,1}, b_{i\,2\,2}, ...], ...\ [b_{i\,n\,1}, b_{i\,n\,2}, ...]\ ]$ where each $b_{i\,x\,y}$ is an integer in the range 0 to *k* inclusive, and LR(*k*) state I has *n* items.

As an example, suppose that LRE state E_3 has the following representation:

$\{\ \langle 17, [\ [1,2], [3], [3]\ ]\ \rangle, \langle 23, [\ [2]\ ]\ \rangle\ \}$

This would mean that state E_3 represents a mixture of the same items as found in the LR(*k*) states numbered 17 and 23. State 17 must have three items (the length of the list that completes the tuple with state number 17) – let us suppose that these items are:

$A \rightarrow A \bullet B\ C \qquad \alpha_1$

$X \rightarrow a\ A \bullet D \qquad \alpha_2$

$A \rightarrow \bullet\ b \qquad \alpha_3$

where we have written the lookahead strings as α_1, α_2 and α_3 respectively. Similarly, LR(*k*) state 23 must have just one item and let us suppose that this item is

$C \rightarrow a\ b \bullet b \qquad \beta_1$

Now, our LRE state represents an Earley state which contains exactly these items:

$\{ \langle A \rightarrow A \bullet B\ C, \alpha_1, 1 \rangle, \langle A \rightarrow A \bullet B\ C, \alpha_1, 2 \rangle, \langle X \rightarrow a\ A \bullet D, \alpha_2, 3 \rangle,$
$\langle A \rightarrow \bullet\ b, \alpha_3, 3 \rangle, \langle C \rightarrow a\ b \bullet b, \beta_1, 2 \rangle \}$

The first tuple in E_3 represents two copies of the first item of LR state 17, where one copy is associated with a pointer back to state 1 and the other with a pointer back to state 2. And similarly for the other items in LR states 17 and 23.

Our parsing algorithm is based on Earley's, but it has been modified to work with the different state representation. It has two main functions named SCAN and RECOGNIZER.

Given a LRE state E_s, the function SCAN(E_s, X, t) constructs a new LRE state which represents Earley items where the marker has been past the token X in all applicable Earley items represented in set E_s.

The procedure RECOGNIZER(x_1, ..., x_n, ..., x_{n+k}) determines whether the token string x_1 ... x_n is in the language generated by G. Note that each of the symbols x_{n+1}, x_{n+2} ... x_{n+k} is the symbol ⊣. These extra *k* symbols are needed to provide right context for the final reductions in the parse. RECOGNIZER constructs a sequence of Earley states, from which a set of valid parse trees may be enumerated. Code for the SCAN function is shown in Figure 1, while code for the RECOGNIZER is given in Figure 2.

The code uses the data structures and tables explained below. The tables may be created during the LR(*k*) parser construction algorithm.

- Each LRE state is represented by a set whose elements are structures with two fields. One field is named `State` and holds a state number for the LR(*k*) recognizer. The other field is named `BackPtrs` and is an array of lists of integers. An element `BackPtrs[i]` holds the state numbers that should be associated with the *i*-th item of LR(*k*) state with number `State`.
- The array `NumberOfItems[i]` gives the number of items in LR(*k*) state *i*.
- The array `SHIFT[s,x]` holds the shift actions for the LR(*k*) recognizer. If the current LR(*k*) state is numbered *s*, then `SHIFT[s,x]` gives the number of the destination state to shift on symbol *x*. If a valid shift action is not defined for symbol *x*, `SHIFT[s,x]` holds -1.
- The array `DestItemPosition[m,i]` gives the correspondence between items in one LR(*k*) state and those in another LR(*k*) state. In particular, if item *i* in the LR(*k*) state numbered *m* is $A \rightarrow \alpha \bullet X\ \beta$, then a shift on the symbol X will lead to a unique destination LR(*k*) state that contains the item $A \rightarrow \alpha\ X \bullet \beta$. The number held in `DestItemPosition[m,i]` is the number of this item in the destination state. If item *i* in state *m* does not have the specified form (i.e. the marker is at the end of the right-hand side), we assume that `DestItemPosition[m,i]` holds the value -1.

```
function SCAN(E_S, X, t)
begin
   result := ∅;
   for origin := each tuple in E_S do
   begin
      dest := Shift[origin.State,X];
      if dest ≥ 0 then
      begin
         newTuple := < dest, emptyBackPtrArray >;
         (* process kernel items of new state *)
         for i := 1 to NumberOfItems[origin] do
         begin
            j := DestItemPosition[origin,i];
            if j ≥ 0 then
               newTuple.BackPtrs[j]:= origin.BackPtrs[i]
         end;
         (* process non-kernel items of new state *)
         for j := 1 to NumberOfItems[dest] do
         begin
            if newTuple.BackPtrs[j] = empty then
               newTuple.BackPtrs[j] := [t]
         end;
         result := MERGE1(result, newTuple)
      end
   end;
   return result
end SCAN;


(* MERGE1 is an auxiliary function called by SCAN *)
function MERGE1( L, T )
begin
   for elem := each element of L do
      if elem.State = T.State then
      begin
         for i := each index of elem.BackPtrs do
            elem.BackPtrs[i] := elem.BackPtrs[i] ∪
                                T.BackPtrs[i];
         return L;
      end;
   return L ∪ { T };
end MERGE1;
```

Fig. 1. The SCAN Function

```
function RECOGNIZER( x_1 ... x_n+k )
begin
   E_0 := { < I_initial, [[0]] > };
   E_1 := SCAN(E_0, ⊢, 1);
   for i = 1 to n do
   begin
      E_i+1 := SCAN(E_i, x_i, i+1);
      repeat
         for LS := each element in E_i+1 do
         begin
            (* process reduce items *)
            rs := ReduceItemList(LS.State, x_i+1 x_i+2 ... x_i+k);
            for i := each element in rs do
            begin
               lhs := LeftHandSymbol[LS.State,i];
               for j := each element in
                           LS.BackPtrs[i] do
                  E_i+1 := MERGE(E_i+1, SCAN(E_j,lhs,i+1));
            end
         end
      until E_i+1 does not change;
      if E_i+1 = ∅ then return failure;
   end
   if E_n+1 = { < I_accept, [[0]] > } then
      return success
   else
      return failure
end RECOGNIZER;


(* MERGE is an auxiliary function used above *)
function MERGE( E1, E2 )
begin
   result := E1;
   for elem := each element in E2 do
      result := MERGE1(result, elem);
   return result;
end MERGE;
```

Fig. 2. The RECOGNIZER Function

- The array `ReduceItemList[m,α]` is a list of the positions of all items in LR(*k*) state *m* where the marker is at the end of the right-hand side and where the lookahead string for these items is α.
- The array `LeftHandSymbol[m,i]` gives the symbol which appears on the left-hand side of the *i*-th item in LR(*k*) state *m*.

5 An Example of Operation

To illustrate the operation of the LRE(*k*) parsing method, we use the ambiguous grammar:

1. E → E + E
2. E → n

This grammar is augmented by the extra rule

0. S → ⊢ E ⊣

For simplicity, we choose *k* = 0. From this grammar, we can derive the LR(0) recognizer which has the states and actions shown below in Table 1. Each shift action is preceded by the symbol which selects that shift action. Because a LR(0) parser does not use lookahead, a reduce action is performed no matter what the next symbol is. The word *any* represents the fact that any symbol selects the specified reduce action. The table contains conflicts, in particular note that state 7 implicitly contains two different actions for the case when the lookahead symbol is +.

Table 1: LR(0) Recognizer for the Example Grammar

State	Item No.	Item	Parse Actions
1	1	[S → • ⊢ E ⊣]	⊢ Shift 2
2	1	[S → ⊢ • E ⊣]	E Shift 3 n Shift 4
	2	[E → • E + E]	
	3	[E → • n]	
3	1	[S → ⊢ E • ⊣]	+ Shift 6 ⊣ Shift 5
	2	[E → E • + E]	
4	1	[E → n •]	*any* Reduce 2
5	1	[S → ⊢ E ⊣ •]	*any* Reduce 0
6	1	[E → E + • E]	E Shift 7 n Shift 4
	2	[E → • E + E]	
	3	[E → • n]	
7	1	[E → E + E •]	+ Shift 6 *any* Reduce 1
	2	[E → E • + E]	

From that LR(0) table we derive the tables shown below in Figure 3. Only the significant entries in the two rectangular arrays, `DestItemPosition` and `LeftHandSymbol` are shown (the missing elements in these arrays should never be accessed). Similarly, only the significant entries in the `Shift` array are shown; if any other element is accessed the result should be -1.

Fig. 3. Tables Used In Parser Example

State	Item No.	DestItem Position	LeftHand Symbol
1	1	1	S
2	1	1	S
2	2	2	E
2	3	1	E
3	1	1	S
3	2	1	E
4	1	-1	E
5	1	-1	S
6	1	1	E
6	2	2	E
6	3	1	E
7	1	-1	E
7	2	1	E

State	Number Of Items	Reduce-ItemList
1	1	[ ]
2	3	[ ]
3	2	[ ]
4	1	[1]
5	1	[1]
6	3	[ ]
7	2	[1]

State	Symbol	Shift
1	⊢	2
2	E	3
2	n	4
3	+	6
3	⊣	5
6	E	7
6	n	4
7	+	6

We now trace the states of the LRE(0) parser on the input string n+n+n. The RECOGNIZER function begins by initializing the set E_0 with the initial LRE state $\{\langle 1,[0] \rangle\}$. It represents item 1 of state 1 in the LR(0) recognizer – indicating that the RHS of the rule S→ ⊢ E ⊣ is to be recognized.

Each numbered step in our trace corresponds to the processing of one input symbol, and begins by showing the LRE state that is computed after seeing that input symbol. An explanation of the state's derivation is provided for the first few steps only.

=== The start of input symbol ⊢ is processed ===

1. $E_1 = \{ \langle 2,[[0], [1], [1]] \rangle \}$. RECOGNIZER called SCAN(E_0, ⊢, 1), which looked

up the action for LR(0) state 1 when the input is ⊢. Thus it created the LRE item ⟨ 2,[[], [], []] ⟩ and then it filled in the back pointers. The list [0] was copied from the origin item, while the two lists containing [1] correspond to completion items.

=== The first input symbol n is now processed ===

2. E_2 = { ⟨ 4, [[1]] ⟩, ⟨ 3, [[0], [1]] ⟩ }. RECOGNIZER called SCAN(E_1, n, 2). The ⟨ 4, [[1]] ⟩ element is created because of the LR(0) action for state 2 when the lookahead symbol is n. The other items are created by RECOGNIZER because item 1 in LR(0) state 4 is a reduce item, and the reduce action is triggered by the next input symbol which is +. The LHS symbol for that item is E, and RECOGNIZER called SCAN(E_1, E, 1) to create the two extra items.

   === The second input symbol + is processed ===

3. E_3 = { ⟨ 6, [[1], [3], [3]] ⟩ }.

   === The input symbol n is processed ===

4. E_4 = { ⟨ 4, [[3]] ⟩, ⟨ 7, [[1], [3]] ⟩, ⟨ 3, [[0], [1]] ⟩ }.

   === The input symbol + is processed ===

5. E_5 = { ⟨ 6, [[1,3], [5], [5]] ⟩ }.

   === The input symbol n is processed ===

6. E_6 = { ⟨ 4, [[5]] ⟩, ⟨ 7, [[1,3], [3,5]] ⟩, ⟨ 3, [[0], [1]] ⟩ }.

   === The end-of-input symbol ⊣ is processed ===

7. E_7 = { ⟨ 5, [0] ⟩ }.

6 An Additional Enhancement

The algorithm presented above can be further improved. The implementation used in our experiments does not immediately record non-kernel items in a LRE state (except when handling productions with an empty RHS). Their processing is deferred until scanning to the next state occurs. By recording the number of kernel items in each LR(*k*) state, and by consulting the `DestItemPosition` table, it can be determined whether or not a particular item in a destination state came from a kernel item in the source state. If it did, the `BackPtr` list is copied from the previous state. If it did not, the list [*t*-1] is supplied, where *t* is the number of the current LRE state.

The additional improvement achieves significant space and time savings, because many predictions items in an Earley parser are fruitless

7 Experimental Results

Lookahead significantly affects the speed of an Earley parser. In general, it is used to eliminate items from the sets of items maintained by the parser. Fewer items imply that fewer fruitless parsing possibilities are explored. On the other hand, a conventional

Earley parser computes the lookahead contexts for items at run-time, and choosing a large value for the lookahead *k* will waste execution time. In Figure 4, we compare the speed of a conventional Earley parser and our LRE parsing method for *k*=0 and *k*=1.

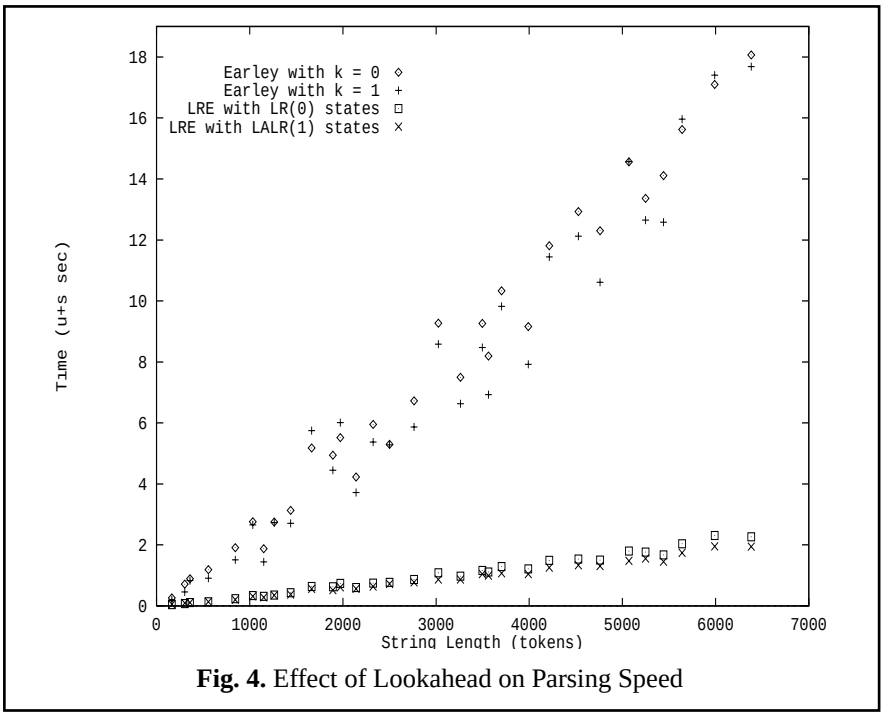

Fig. 4. Effect of Lookahead on Parsing Speed

Figure 4 already demonstrates that LRE(*k*) is a much faster parsing method than the conventional Earley parsing method. In Figure 5, we show an additional comparison against a parser generated by the freely distributed parser generator *bison* [1]. (Other measurements, not displayed here, reveal that a parser generated by *yacc* [5] yields very similar results.) Our grammar for these experiments was Roskind's ANSI C grammar. The grammar contains one ambiguity, namely the dangling *else* problem. This ambiguity is automatically eliminated from the generated parser when *yacc* and *bison* are used; it is retained by the Earley parsers.

For an unambiguous grammar (or when the ambiguities have been eliminated, such as with the *bison*'s interpretation of the Roskind C grammar), recognition time is proportional to the length of the input. For an ambiguous grammar, the recognition time may increase as the cube of the length of the input. Figure 6 shows timing measurements when parsing with the ambiguous grammar:

$$S \rightarrow S\ S \mid a,$$

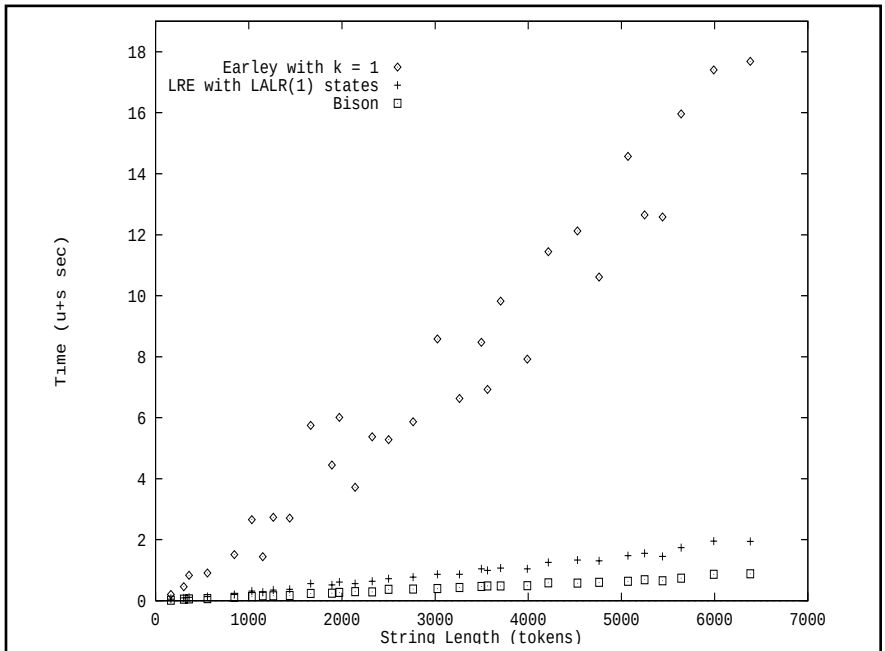

Fig. 5. Parser Speed for Roskind C Grammar

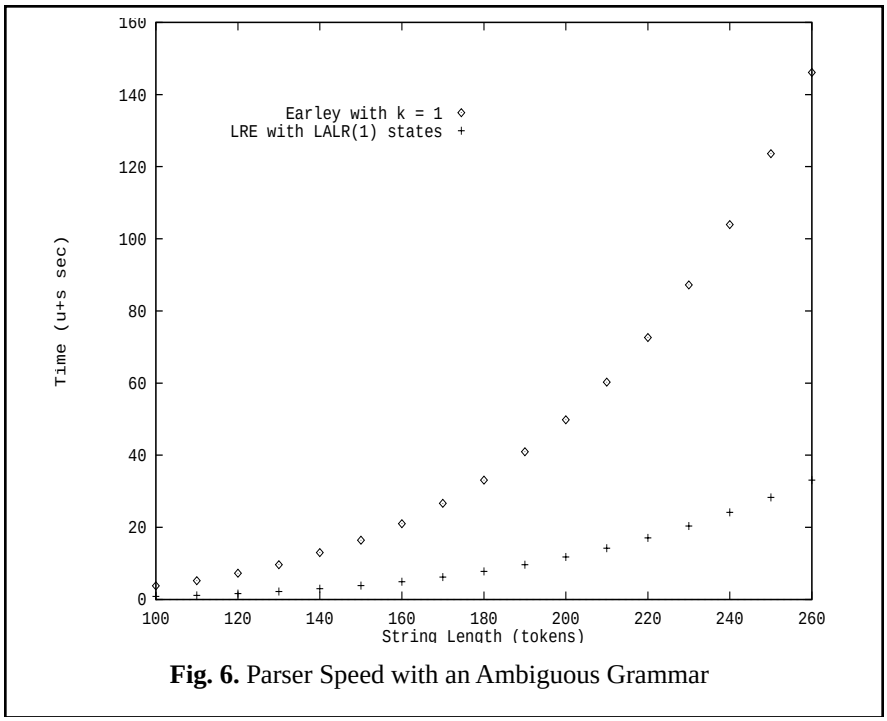

Fig. 6. Parser Speed with an Ambiguous Grammar

8 Conclusions

We have modified Earley's parsing method so that it can take advantage of precomputed LR(k) sets of items. The result is a hybrid parsing method, LRE(k), which can still handle general context-free grammars but which is comparable in speed to a *yacc*-generated or *bison*-generated parser. However, *yacc* and *bison* can, of course, only recognize unambiguous languages that are based on LALR(1) grammars with conflict elimination in the generated parser. The LRE(k) parsing method is 10 to 15 times faster than a conventional Earley parser, while requiring less than half the storage.

Acknowledgements

Funding for this research was provided by the Natural Sciences and Engineering Research Council of Canada in the form of a summer fellowship for the first author and a research grant for the second author. The initial motivation for working on this problem is due to Gordon Cormack.

References

1. Donnelly, C., and Stallman, R. *BISON: Reference Manual*. Free Software Foundation, Cambridge, MA, 1992 .
2. Earley, J. *An Efficient Context-Free Parsing Algorithm*. Comm. ACM **13, 2** (Feb. 1970), 94-102.
3. Earley, J. *An Efficient Context-Free Parsing Algorithm*. Ph.D. Thesis, Carnegie-Mellon University, 1968.
4. Grune, D., and Jacob, C.J.H. *Parsing Techniques: a practical guide*. Ellis Horwood, Chichester, 1990.
5. Johnson, S.C. *YACC: Yet Another Compiler-Compiler*. UNIX Programmer's Supplementary Documents, vol 1, 1986.
6. Tomita, M. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston, 1986.