Directly-Executable Earley Parsing

John Aycock and Nigel Horspool

Department of Computer Science,
University of Victoria,
Victoria, B. C., Canada V8W 3P6
{aycock,nigelh}@csc.uvic.ca

Abstract. Deterministic parsing techniques are typically used in favor of general parsing algorithms for efficiency reasons. However, general algorithms such as Earley's method are more powerful and also easier for developers to use, because no seemingly arbitrary restrictions are placed on the grammar. We describe how to narrow the performance gap between general and deterministic parsers, constructing a directly-executable Earley parser that can reach speeds comparable to deterministic methods even on grammars for commonly-used programming languages.

1 Introduction

Most parsers in use today are only capable of handling subsets of context-free grammars: LL, LR, LALR. And with good reason – efficient linear-time algorithms for parsing these subsets are known. In contrast, general parsers which can handle any context-free grammar are slower due to extra overhead, even for those cases where the general algorithm runs in linear time [13].

However, general algorithms have some advantages. No "massaging" of a context-free grammar is required to make it acceptable for use in a general parser, as is required by more efficient algorithms like the LALR(1) algorithm used in Yacc [15]. Using a general parser thus reduces programmer development time, eliminates a source of potential bugs, and lets the grammar reflect the input language rather than the limitations of a compiler tool.

General algorithms also work for ambiguous grammars, unlike their more efficient counterparts. Some programming language grammars, such as those for Pascal, C, and C++, contain areas of ambiguity. For some tasks ambiguous grammars may be deliberately constructed, such as a grammar which describes multiple dialects of a language for use in software reengineering [7].

The primary objection to general parsing algorithms, then, is not one of functionality but of speed. For LR parsers, dramatic speed improvements have been obtained by producing hard-coded, or directly-executable parsers [3, 5, 14, 23, 24]. These directly-executable LR parsers implement the parser for a given grammar as a specialized program, rather than using a typical table-driven approach. In this paper we extend directly-executable techniques for use in Earley's general parsing algorithm, to produce and evaluate what we believe is the first directly-executable Earley parser. The speed of our parsers is shown to be comparable to deterministic parsers produced by Bison.

2 Earley Parsing

Earley's algorithm [9, 10] is a general parsing algorithm; it can recognize input described by any context-free grammar (CFG). (We assume the reader is familiar with the standard definition and notation of CFGs.) As in [1], uppercase letters (A, B) represent nonterminals, lowercase letters (a, b) represent terminals, and α and β are strings of terminal and nonterminal symbols. Also, every CFG G is augmented by adding a new start rule $S' \to S$, where S is the original start symbol of G.

Earley's algorithm works by building a sequence of Earley sets,[1] one initial Earley set S_0, and one Earley set S_i for each input symbol x_i. An Earley set contains Earley items, which consist of three parts: a grammar rule; a position in the grammar rule's right-hand side indicating how much of that rule has been seen, denoted by a dot ($\bullet$); a pointer back to some previous "parent" Earley set. For instance, the Earley item $[A \to a \bullet Bb, 12]$ indicates that the parser has seen the first symbol of the grammar rule $A \to aBb$, and points back to Earley set S_{12}. We use the term "core Earley item" to refer to an Earley item less its parent pointer: $A \to a \bullet Bb$ in the above example.

The three steps below are applied to Earley items in S_i until no more can be added; this constructs S_i and primes S_{i+1}.

SCANNER. If $[A \to \cdots \bullet b \cdots, j]$ is in S_i and $x_i = b$, add $[A \to \cdots b \bullet \cdots, j]$ to S_{i+1}.

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

COMPLETER. If a "final" Earley item $[A \to \cdots \bullet, j]$ is in S_i, add $[B \to \cdots A \bullet \cdots, k]$ to S_i for all Earley items $[B \to \cdots \bullet A \cdots, k]$ in S_j.

An Earley item is added to a Earley set only if it is not already present in the Earley set. The initial set S_0 holds the single Earley item $[S' \to \bullet S, 0]$ prior to Earley set construction, and the final Earley set must contain $[S' \to S\bullet, 0]$ upon completion in order for the input string to be accepted. For example, Fig. 1 shows the Earley sets when parsing an input using the expression grammar G_E:

$$\begin{array}{lll} S' \to E & T \to T * F & F \to n \\ E \to E + T & T \to T / F & F \to -F \\ E \to E - T & T \to F & F \to +F \\ E \to T & & F \to (E) \end{array}$$

[1] To avoid confusion later, we use the unfortunately awkward terms "Earley set" and "Earley item" throughout.

	`n`	`+`	`n`
S_0	S_1	S_2	S_3
$S' \to \bullet E \quad ,0$	$\mathbf{F \to n\bullet} \quad ,\mathbf{0}$	$E \to E + \bullet T \ ,0$	$\mathbf{F \to n\bullet} \quad ,\mathbf{2}$
$E \to \bullet E + T \ ,0$	$\mathbf{T \to F\bullet} \quad ,\mathbf{0}$	$T \to \bullet T * F \ ,2$	$\mathbf{T \to F\bullet} \quad ,\mathbf{2}$
$E \to \bullet E - T \ ,0$	$\mathbf{E \to T\bullet} \quad ,\mathbf{0}$	$T \to \bullet T/F \ ,2$	$\mathbf{E \to E + T\bullet} \ ,\mathbf{0}$
$E \to \bullet T \quad ,0$	$T \to T \bullet * F \ ,0$	$T \to \bullet F \quad ,2$	$T \to T \bullet * F \ ,2$
$T \to \bullet T * F \ ,0$	$T \to T \bullet /F \ ,0$	$F \to \bullet n \quad ,2$	$T \to T \bullet /F \ ,2$
$T \to \bullet T/F \ ,0$	$S' \to E\bullet \quad ,0$	$F \to \bullet - F \ ,2$	$\mathbf{S' \to E\bullet} \quad ,\mathbf{0}$
$T \to \bullet F \quad ,0$	$E \to E \bullet + T \ ,0$	$F \to \bullet + F \ ,2$	
$F \to \bullet n \quad ,0$	$E \to E \bullet - T \ ,0$	$F \to \bullet (E) \ ,2$	
$F \to \bullet - F \ ,0$			
$F \to \bullet + F \ ,0$			
$F \to \bullet (E) \ ,0$			

Fig. 1. Earley sets for the expression grammar G_E, parsing the input `n + n`. Emboldened final Earley items are ones which correspond to the input's derivation.

The Earley algorithm may employ lookahead to reduce the number of Earley items in each Earley set, but we have found the version of the algorithm without lookahead suitable for our purposes. We also restrict our attention to input recognition rather than parsing proper. Construction of parse trees in Earley's algorithm is done after recognition is complete, based on information retained by the recognizer, so this division may be done without loss of generality.

There are a number of observations about Earley's algorithm which can be made. By themselves, they seem obvious, yet taken together they shape the construction of our directly-executable parser.

Observation 1. Additions are only ever made to the current and next Earley sets, S_i and S_{i+1}.

Observation 2. The Completer does not recursively look back through Earley sets; it only considers a single parent Earley set, S_j.

Observation 3. The Scanner looks at each Earley item exactly once, and this is the only place where the dot may be moved due to a terminal symbol.

Observation 4. Earley items added by Predictor all have the same parent, i.

3 DEEP: a Directly-Executable Earley Parser

3.1 Basic Organization

The contents of an Earley set depend on the input and are not known until run time; we cannot realistically precompute one piece of directly-executable code for every possible Earley set. We can assume, however, that the grammar is known prior to run time, so we begin by considering how to generate one piece of directly-executable code per Earley item.

Even within an Earley item, not everything can be precomputed. In particular, the value of the parent pointer will not be known ahead of time. Given two otherwise identical Earley items $[A \to \alpha \bullet \beta, j]$ and $[A \to \alpha \bullet \beta, k]$, the invariant part is the core Earley item. The code for a directly-executable Earley item, then, is actually code for the core Earley item; the parent pointer is maintained as data. A directly-executable Earley item may be represented as the tuple

$$(\textit{code for } A \to \alpha \bullet \beta,\ \textit{parent})$$

the code for which is structured as shown in Fig. 2. Each terminal and nonterminal symbol is represented by a distinct number; the variable `sym` can thus contain either type of symbol. Movement over a terminal symbol is a straightforward implementation of the SCANNER step, but movement over a nonterminal is complicated by the fact that there are two actions that may take place upon reaching an Earley item $[A \to \cdots \bullet B \cdots, j]$, depending on the context:

1. If encountered when processing the current Earley set, the PREDICTOR step should be run.
2. If encountered in a parent Earley set (i.e., the COMPLETER step is running) then movement over the nonterminal may occur. In this case, `sym` cannot be a terminal symbol, so the predicate `ISTERMINAL()` is used to distinguish these two cases.

The code for final Earley items calls the code implementing the parent Earley set, after replacing `sym` with the nonterminal symbol to move the dot over. By Observation 2, no stack is required as the call depth is limited to one. Again, this should only be executed if the current set is being processed, necessitating the `ISTERMINAL()`.

3.2 Earley Set Representation

An Earley set in the directly-executable representation is conceptually an ordered sequence of *(code, parent)* tuples followed by one of the special tuples:

$$(\textit{end of current Earley set code},\ -1)$$
$$(\textit{end of parent Earley set code},\ -1)$$

The code at the end of a parent Earley set simply contains a `return` to match the `call` made by a final Earley item. Reaching the end of the current Earley set is complicated by bookkeeping operations to prepare for processing the next Earley set. The parent pointer is irrelevant for either of these.

In practice, our DEEP implementation splits the tuples. DEEP's Earley sets are in two parts: a list of addresses of code, and a corresponding list of parent pointers. The two parts are separated in memory by a constant amount, so that knowing the memory location of one half of a tuple makes it a trivial calculation to find the other half.

Having a list of code addresses for an Earley set makes it possible to implement the action "`goto next Earley item`" with direct threaded code [4]. With threaded code, each directly-executable Earley item jumps directly to the beginning of the code for the next Earley item, rather than first returning to

$[A \rightarrow \cdots \bullet a \cdots, j]$ (movement over a terminal) $\Rightarrow$

```
if (sym == a) {
    add [A → ··· a • ···, j] to S_{i+1}
}
goto next Earley item
```

$[A \rightarrow \cdots \bullet B \cdots, j]$ (movement over a nonterminal) $\Rightarrow$

```
if (ISTERMINAL(sym)) {
    foreach rule B → α {
        add [B → • α, i] to S_i
    }
} else if (sym == B) {
    add [A → ··· B • ···, j] to S_i
}
goto next Earley item
```

$[A \rightarrow \cdots \bullet, j]$ (final Earley item) $\Rightarrow$

```
if (ISTERMINAL(sym)) {
    saved_sym = sym
    sym = A
    call code for Earley set S_j
    sym = saved_sym
}
goto next Earley item
```

Fig. 2. Pseudocode for directly-executable Earley items.

a dispatching loop or incurring function call overhead. Not only does threaded code proffer speed advantages [16], but it can work better with branch prediction hardware on modern CPUs [11]. We implement this in a reasonably portable fashion using the first-class labels in GNU C.[2]

How is an Earley item added to an Earley set in this representation? First, recall that an Earley item is only placed in an Earley set if it does not already appear there. We will use the term "appending" to denote an Earley item being placed into an Earley set; "adding" is the process of determining if an Earley item should be appended to an Earley set. (We discuss adding more in Section 3.3.) Using this terminology, appending an Earley item to an Earley set is done by dynamically generating threaded code. We also dynamically modify the threaded code to exchange one piece of code for another in two instances:

1. When the current Earley set is fully processed, "end of current Earley set" must be exchanged for "end of parent Earley set."
2. Observation 3 implies that once the SCANNER has looked at an Earley item, any code looking for terminal symbols is superfluous. By modifying the threaded code, DEEP skips the superfluous code on subsequent invocations.

Appending leaves DEEP with a thorny problem of memory management. Observation 1 says that Earley items – threaded code plus separate parent pointers – can be appended to one of two Earley sets. We also maintain an array whose

[2] This is an extension to ANSI C.

i^{th} entry is a pointer to the code for Earley set S_i, for implementation of `call`, giving us a total of five distinct, dynamically-growing memory areas.

Instead of complex, high-overhead memory management, we have the operating system assist us by memory-mapping oversized areas of virtual memory. This is an efficient operation because the operating system will not allocate the virtual memory pages until they are used. We can also protect key memory pages so that an exception is caused if DEEP should exceed its allocated memory, absolving us from performing bounds checking when appending. This arrangement is shown in Fig. 3, which also demonstrates how the current and next Earley sets alternate between memory areas.

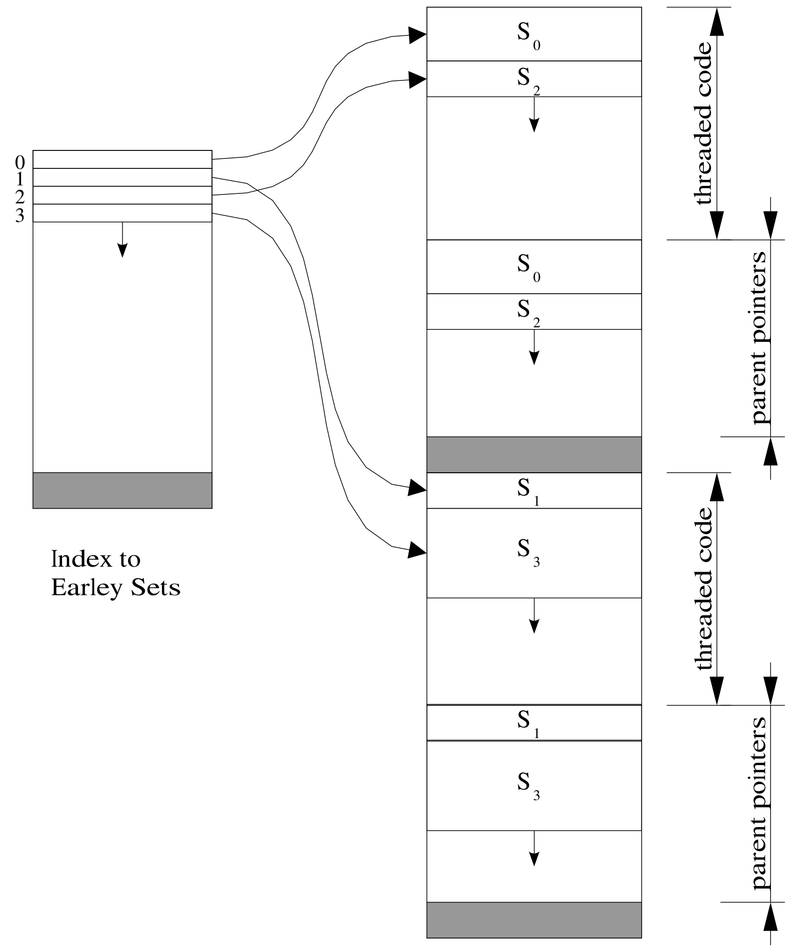

Fig. 3. Memory layout for DEEP. S_2 and S_3 are the current and next Earley sets, respectively; the shaded areas are protected memory pages.

3.3 Adding Earley Items

As mentioned, adding an Earley item to an Earley set entails checking to ensure that it is not already present. Earley suggested using an array indexed by the parent pointer, each entry of which would be a list of Earley items to search [10]. Instead, we note that core Earley items may be enumerated, yielding finite, relatively small numbers.[3] A core Earley item's number may be used to index into a bitmap to quickly check the presence or absence of *any* Earley item with that core.

When two or more Earley items exist with the same core, but different parent pointers, we construct a radix tree [17] for that core Earley item – a binary tree whose branches are either zero or one – which keeps track of which parent pointers have been seen. Radix trees have two nice properties:

1. Insertion and lookup, the only operations required, are simple.
2. The time complexity of radix tree operations during execution is $\log i$, where i is the number of tokens read, thus growing slowly even with large inputs.

To avoid building a radix tree for a core Earley item until absolutely necessary, we cache the first parent pointer until we encounter a second Earley item with the same core. An example of adding Earley items is given in Fig. 4.

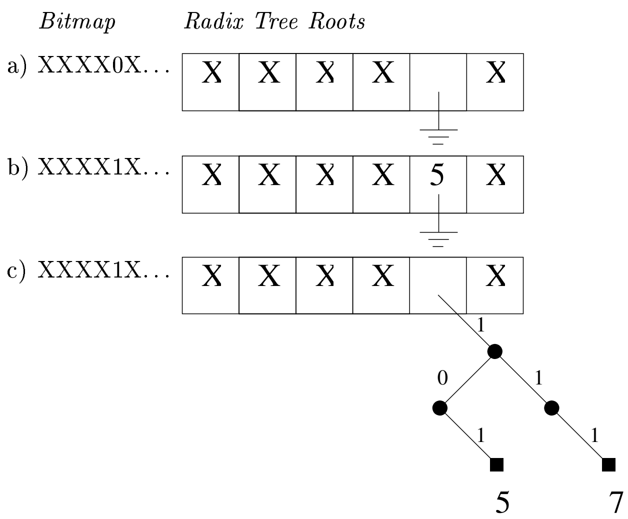

Fig. 4. Adding Earley items: (a) initial state; (b) after appending Earley item #4, parent S_5; (c) after appending Earley item #4, parent S_7. In the radix tree, a circle (•) indicates an absent entry, and a square (■) indicates an existing entry. "X" denotes a "don't care" entry.

[3] To be precise, the number of core Earley items is $\sum_{A \to \alpha \in G}(|\alpha| + 1)$.

3.4 Sets Containing Items which are Sets Containing Items

Earley parsing has a deep relationship with its contemporary, LR parsing [9]. Here we look at LR(0) parsers – LR parsers with no lookahead. As with all LR parsers, an LR(0) parser's recognition is driven by a deterministic finite automaton (DFA) which is used to decide when the right-hand side of a grammar rule has been seen. A DFA state corresponds to a set of LR(0) items, and an LR(0) item is exactly the same as a core Earley item.

How is an LR(0) DFA constructed? Consider a *non*deterministic finite automaton (NFA) for LR(0) parsing, where each NFA state contains exactly one LR(0) item. A transition is made from $[A \to \cdots \bullet X \cdots]$ to $[A \to \cdots X \bullet \cdots]$ on the symbol X, and from $[A \to \cdots \bullet B \cdots]$ to $[B \to \bullet\alpha]$ on ϵ; the start state is $[S' \to \bullet S]$. This NFA may then be converted into the LR(0) DFA using standard methods [1].

The conversion from NFA to DFA yields, as mentioned, DFA states which are sets of LR(0) items. Within each LR(0) set, the items may be logically divided into kernel items (the initial item and items where the dot is not at the beginning) and nonkernel items (all other items) [1]. We explicitly represent this logical division by splitting each LR(0) DFA state into two states (at most), leaving us with an almost-deterministic automaton, the LR(0) $\overline{\text{DFA}}$. Figure 5 shows a partial LR(0) $\overline{\text{DFA}}$ for G_E (the remainder is omitted due to space constraints). In the original LR(0) DFA, states 0 and 1, 2 and 10, 18 and 19, 24 and 25 were merged together.

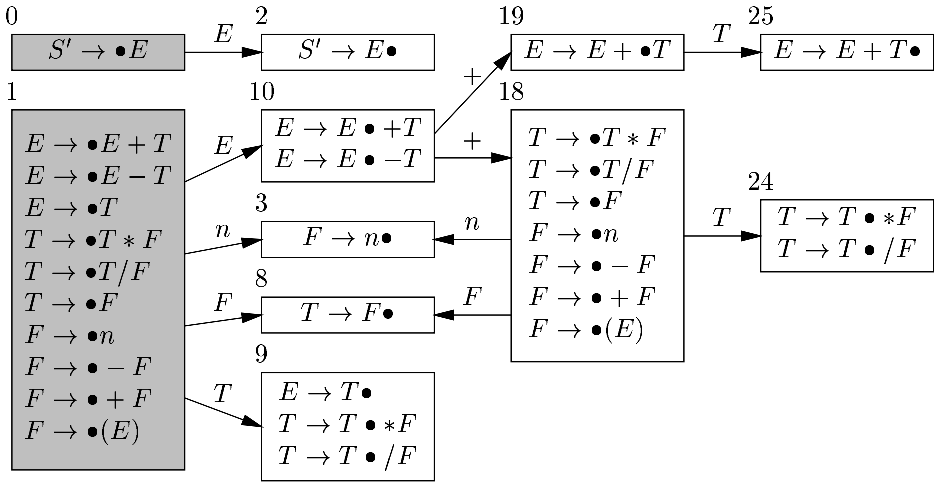

Fig. 5. Partial LR(0) $\overline{\text{DFA}}$ for G_E. Shading denotes start states.

Returning to Earley parsing, the core Earley items in an Earley set may be represented using one or more states in an LR(0) DFA [22]. The problem with doing so is that keeping track of which parent pointers and LR(0) items belong together results in a complex, inelegant implementation. However, we realized as a result of Observation 4 that the PREDICTOR really just corresponds to making a transition to a "nonkernel" state in the LR(0) $\overline{\text{DFA}}$. Pursuing this idea, we represent Earley items in DEEP as the tuples

$$(\textit{code for LR(0) } \overline{\textit{DFA}} \textit{ state, parent})$$

Figure 6 shows the Earley sets from Fig. 1 recoded using the LR(0) $\overline{\text{DFA}}$ states.

	`n`	`+`	`n`
S_0	S_1	S_2	S_3
0 , 0	3 , 0	19 , 0	3 , 2
1 , 0	8 , 0	18 , 2	8 , 2
	9 , 0		24 , 2
	10 , 0		25 , 0
	2 , 0		2 , 0

Fig. 6. Earley sets for the expression grammar G_E, parsing the input `n + n`, encoded using LR(0) $\overline{\text{DFA}}$ states.

Through this new representation, we gain most of the efficiency of using an LR(0) DFA as the basis of an Earley parser, but with the benefit of a particularly simple representation and implementation. The prior discussion in this section regarding DEEP still holds, except the directly-executable code makes transitions from one LR(0) $\overline{\text{DFA}}$ state to another instead of from one Earley item to another.

4 Evaluation

We compared DEEP with three different parsers:

1. ACCENT, an Earley parser generator [25].
2. A standard implementation of an Earley parser, by the second author.
3. Bison, the GNU incarnation of Yacc.

All parsers were implemented in C, used the same (`flex`-generated) scanner, and were compiled with `gcc` version 2.7.2.3 using the `-O` flag. Timings were conducted on a 200 MHz Pentium with 64 M of RAM running Debian GNU/Linux version 2.1.

Figure 7 shows the performance of all four on G_E, the expression grammar from Sect. 2. As expected, Bison is the fastest, but DEEP is a close second, markedly faster than the other Earley parsers.

In Fig. 8 the parsers (less Bison) operate on an extremely ambiguous grammar. Again, DEEP is far faster than the other Earley parsers. The performance curves themselves are typical of the Earley algorithm, whose time complexity is $O(n)$ for most LR(k) grammars, $O(n^2)$ for unambiguous grammars, and $O(n^3)$ in the worst case.[10]

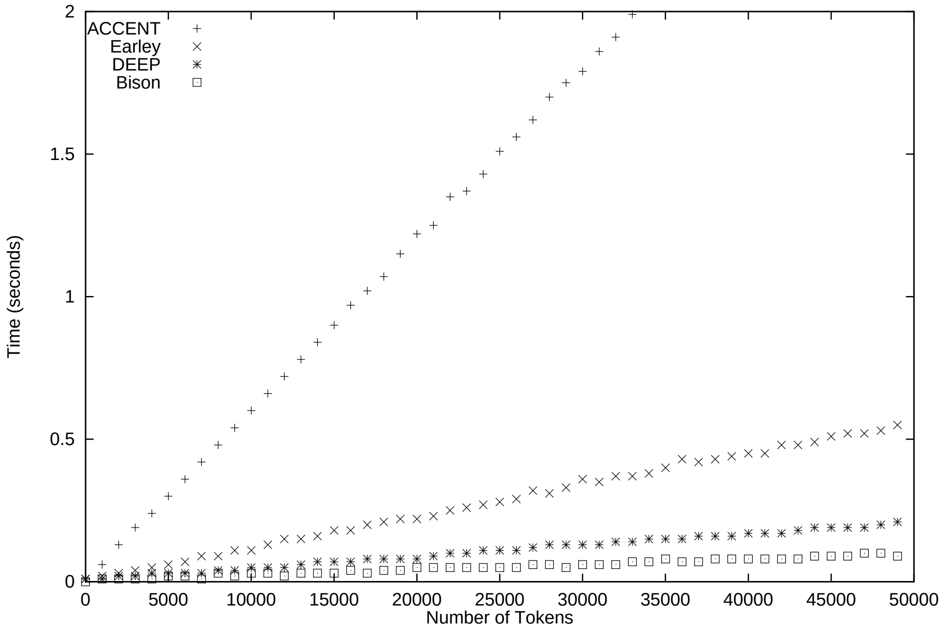

Fig. 7. Timings for the expression grammar, G_E.

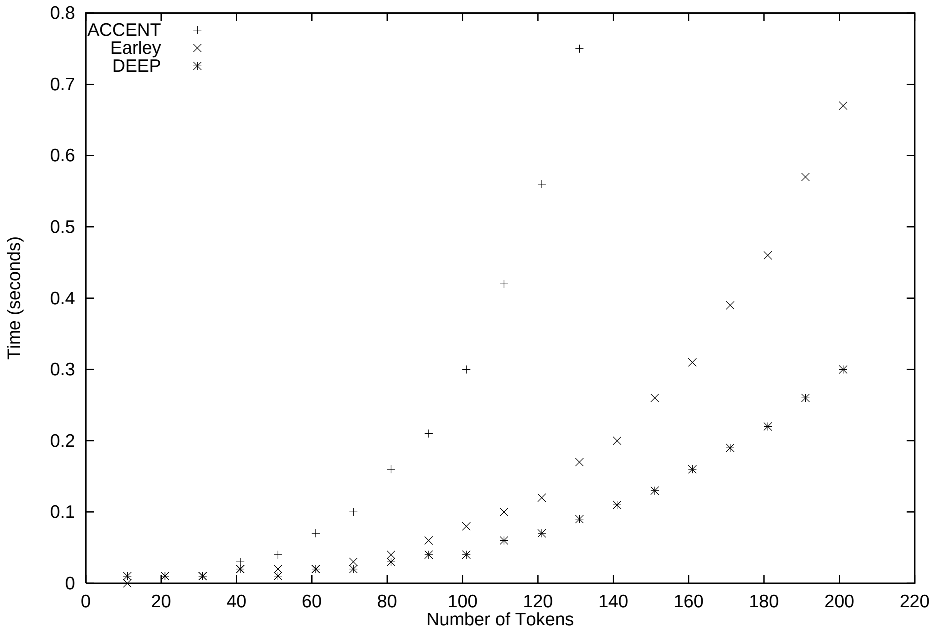

Fig. 8. Timings for the ambiguous grammar $S \to SSx|x$.

5 Improvements

Next, we tried DEEP on the Java 1.1 grammar [12] which consists of 350 grammar rules.[4] Suffice it to say that only the extremely patient would be content to wait while `gcc` compiled and optimized this monster. To make DEEP practical, its code size had to be reduced.

Applying Observation 3, code looking for terminal symbols may only be executed once during the lifetime of a given Earley item. In contrast, an Earley item's nonterminal code may be invoked many times. To decrease the code size, we excised the directly-executable code for terminal symbols, replacing it with a single table-driven interpreter which interprets the threaded code. Nonterminal code is still directly-executed, when COMPLETER calls back to a parent Earley set. This change reduced compile time by over 90%.

Interpretation allowed us to trivially make another improvement, which we call "Earley set compression." Often, states in the LR(0) $\overline{\text{DFA}}$ have no transitions on nonterminal symbols; the corresponding directly-executable code is a no-op which can consume both time and space. The interpreter looks for such cases and removes those Earley items, since they cannot contribute to the parse. We think of Earley set compression as a space optimization, because only a negligible performance improvement resulted from its implementation.

The version of DEEP using partial interpretation and Earley set compression is called SHALLOW. Figure 9 compares the performance of SHALLOW and Bison on realistic Java inputs. SHALLOW can be two to five times slower, although the difference amounts to only fractions of a second – a difference unlikely to be noticed by end users!

One typical improvement to Earley parsers is the use of lookahead. Earley suggested that the COMPLETER should employ lookahead, but this was later shown to be a poor choice [8]. Instead, it was demonstrated that the use of one-token lookahead by the PREDICTOR yielded the best results [8]. This "prediction lookahead" avoids placing Earley items into an Earley set that are obvious dead ends, given the subsequent input. However, the LR(0) $\overline{\text{DFA}}$ naturally clusters together PREDICTOR's output. Where prediction lookahead would avoid generating many Earley items in a standard Earley parser, it would only avoid one Earley item in SHALLOW if all the predicted dead ends fell within a single LR(0) $\overline{\text{DFA}}$ state.

We instrumented SHALLOW to track Earley items that were unused, in the sense that they never caused more Earley items to be added to any Earley set, and were not final Earley items. Averaging over the Java corpus, 16% of the Earley items were unused. Of those, prediction lookahead could remove *at most* 19%; Earley set compression removed 76% of unused Earley items in addition to pruning away other Earley items. We conjecture that prediction lookahead is of limited usefulness in an Earley parser using any type of LR automaton.

[4] This number refers to grammar rules in Backus-Naur form, obtained after transforming the grammar from [12] in the manner they prescribe.

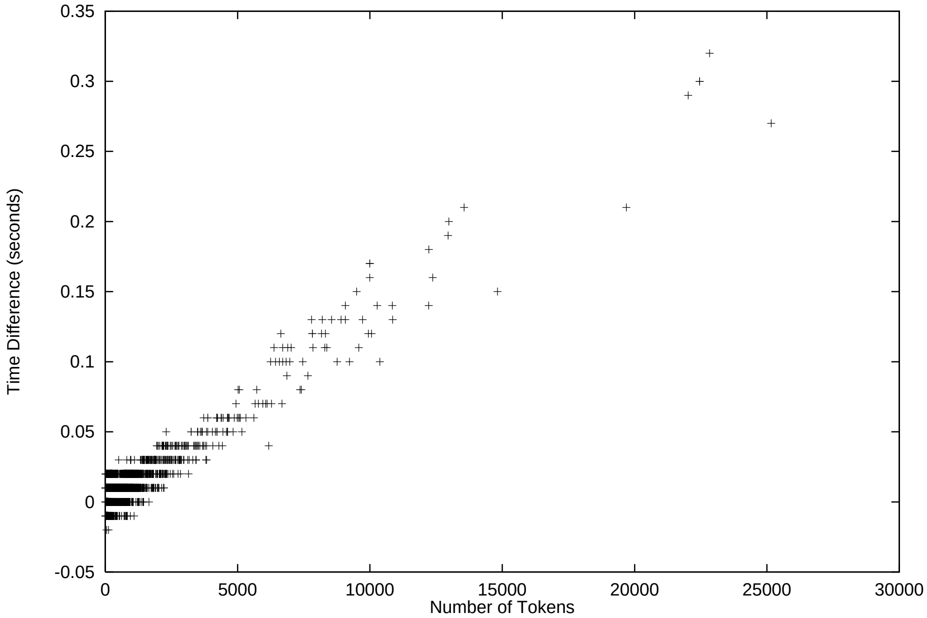

Fig. 9. Difference between SHALLOW and Bison timings for Java 1.1 grammar, parsing 3350 Java source files from JDK 1.2.2 and Java Cup v10j.

6 Related Work

Appropriately, the first attempt at direct execution of an Earley parser was made by Earley himself [9]. For a subset of the CFGs which his algorithm recognized in linear time, he proposed an algorithm to produce a hardcoded parser. Assuming the algorithm worked and scaled to practically-sized grammars – Earley never implemented it – it would only work for a subset of CFGs, and it possessed unresolved issues with termination.

The only other reference to a directly-executable “Earley” parser we have found is Leermakers’ recursive ascent Earley parser [19–21]. He provides a heavily-recursive functional formulation which, like Earley’s proposal, appears not to have been implemented. Leermakers argues that the directly-executable LR parsers which influenced our work are really just manifestations of recursive ascent parsers [20], but he also notes that he uses “recursive ascent Earley parser” to denote parsers which are not strictly Earley ones [21, page 147]. Indeed, his algorithm suffers from a problem handling cyclic grammar rules, a problem not present in Earley’s algorithm (and consequently not present in our Earley parsers).

Using deterministic parsers as an efficient basis for general parsing algorithms was suggested by Lang in 1974 [18]. However, none of the applications of this idea in Earley parsers [22] and Earley-like parsers [2, 6, 26] have explored the benefits of using an almost-deterministic automaton and exploiting Earley's ability to simulate nondeterminism.

7 Future Work

By going from DEEP to SHALLOW, we arrived at a parser suited for practical use. This came at a cost, however: as shown in Fig. 10, the partially-interpreted SHALLOW is noticeably slower than the fully-executable DEEP. Even with the slowdown, SHALLOW's timings are still comparable to Bison's. One area of future work is determining how to retain DEEP's speed while maintaining the practicality of SHALLOW.

On the other side of the time/space coin, we have yet to investigate ways to reduce the size of generated parsers. Comparing the sizes of stripped executables, SHALLOW parsers for G_E and Java were 1.5 and 9.0 times larger, respectively, than the corresponding Bison parsers.

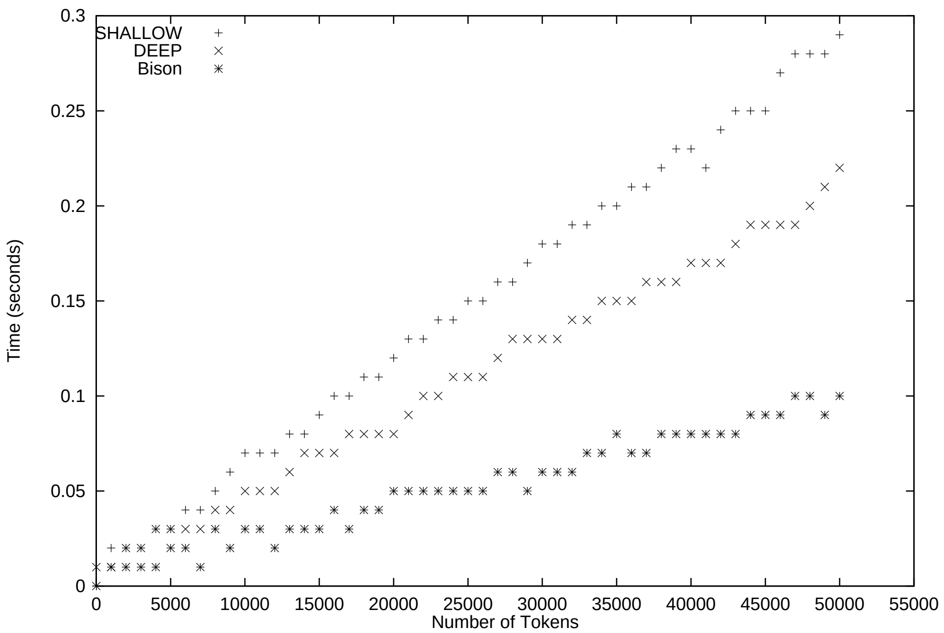

Fig. 10. Performance impact of partial interpretation of G_E.

Additionally, we have not yet explored the possibility of using optimizations based on grammar structure. One such example is elimination of unit rules,[5]

[5] Also called chain rule elimination.

grammar rules such as $A \to B$ with only a single nonterminal on the right-hand side [13]. Techniques like this have been employed with success in other directly-executable parsers [14, 24].

8 Conclusion

We have shown that directly-executable LR parsing techniques can be extended for use in general parsing algorithms such as Earley's algorithm. The result is a directly-executable Earley parser which is substantially faster than standard Earley parsers, to the point where it is comparable with LALR(1) parsers produced by Bison.

9 Acknowledgments

This work was supported in part by a grant from the National Science and Engineering Research Council of Canada. Shannon Jaeger made a number of helpful comments on a draft of this paper.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. M. A. Alonso, D. Cabrero, and M. Vilares. Construction of Efficient Generalized LR Parsers. *Proceedings of the Second International Workshop on Implementing Automata*, 1997, pp. 131–140.
3. D. T. Barnard and J. R. Cordy. SL Parses the LR Languages. *Computer Languages 13*, 2 (1988), pp. 65–74.
4. J. R. Bell. Threaded Code. *CACM 16*, 6 (June 1973), pp. 370–372.
5. A. Bhamidipaty and T. A. Proebsting. Very Fast YACC-Compatible Parsers (For Very Little Effort). *Software: Practice and Experience 28*, 2 (February 1998), pp. 181–190.
6. S. Billot and B. Lang. The Structure of Shared Forests in Ambiguous Parsing. *Proceedings of the 27^{th} Annual Meeting of the Association for Computational Linguistics*, 1989, pp. 143–151.
7. M. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. *International Workshop on Program Comprehension*, 1998, pp. 108–117.
8. M. Bouckaert, A. Pirotte, and M. Snelling. Efficient Parsing Algorithms for General Context-free Parsers. *Information Sciences 8*, 1975, pp. 1–26.
9. J. Earley. *An Efficient Context-Free Parsing Algorithm*, Ph.D. thesis, Carnegie-Mellon University, 1968.
10. J. Earley. An Efficient Context-Free Parsing Algorithm. *CACM 13*, 2 (February 1970), pp. 94–102.
11. M. A. Ertl and D. Gregg. Hardware Support for Efficient Interpreters: Fast Indirect Branches (Draft). May, 2000.
12. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

13. D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.
14. R. N. Horspool and M. Whitney. Even Faster LR Parsing. *Software: Practice and Experience 20*, 6 (June 1990), pp. 515–535.
15. S. C. Johnson. Yacc: Yet Another Compiler-Compiler. *Unix Programmer's Manual* (7^{th} edition), volume 2B, 1978.
16. P. Klint. Interpretation Techniques. *Software: Practice and Experience 11*, 1981, pp. 963–973.
17. D. E. Knuth. *The Art of Computer Programming Volume 3: Sorting and Searching* (2^{nd} edition), Addison-Wesley, 1998.
18. B. Lang. Deterministic Techniques for Efficient Non-Deterministic Parsers. In *Automata, Languages, and Programming (LNCS #14)*, J. Loeckx, ed., Springer-Verlag, 1974.
19. R. Leermakers. A recursive ascent Earley parser. *Information Processing Letters 41*, 1992, pp. 87–91.
20. R. Leermakers. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science 104*, 1992, pp. 299–312.
21. R. Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic, 1993.
22. P. McLean and R. N. Horspool. A Faster Earley Parser. *Proceedings of the International Conference on Compiler Construction, CC '96*, 1996, pp. 281–293.
23. T. J. Pennello. Very Fast LR Parsing. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, *SIGPLAN 21*, 7 (1986), pp. 145–151.
24. P. Pfahler. Optimizing Directly Executable LR Parsers. *Compiler Compilers, Third International Workshop, CC '90*, 1990, pp. 179–192.
25. F. W. Schröer. *The ACCENT Compiler Compiler, Introduction and Reference*. GMD Report 101, German National Research Center for Information Technology, June 2000.
26. M. Vilares Ferro and B. A. Dion. Efficient Incremental Parsing for Context-Free Languages. *Proceedings of the 5^{th} IEEE International Conference on Computer Languages*, 1994, pp. 241–252.