Parsing Reflective Grammars*

Paul Stansifer
pauls@ccs.neu.edu

Mitchell Wand
wand@ccs.neu.edu

College of Computer and Information Science
Northeastern University
360 Huntington Avenue
Boston, Massachusetts 02215

ABSTRACT

Existing technology can parse arbitrary context-free grammars, but only a single, static grammar per input. In order to support more powerful syntax-extension systems, we propose reflective grammars, which can modify their own syntax during parsing. We demonstrate and prove the correctness of an algorithm for parsing reflective grammars. The algorithm is based on Earley's algorithm, and we prove that it performs asymptotically no worse than Earley's algorithm on ordinary context-free grammars.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Parsing

General Terms

Languages

Keywords

Earley parsing, context-free languages, syntax extension

1. INTRODUCTION

A software project may involve many different languages with different purposes and complexities, each with its own "natural" syntax. Typically, these languages are segregated from each other, either appearing in separate files, or inside strings. But parenthesis-structured languages from the Lisp family support incremental syntax extension (via macro systems). This extension process provides powerful integration, but the surface syntax is restricted to S-expressions.

We believe it is possible to bridge this gap and create macro systems with the syntactic power of arbitrary context-free grammars. However, new parsing technology is needed to do so. In this paper, we propose reflective grammars, which allow a language designer to define an incrementally extensible base language. In such a language, a valid sentence may contain strings matching productions dynamically added by the sentence itself. This happens in a structured fashion. Users of this language can use its extension construct to write in any surface syntax they want.

These language extensions are dynamic in the sense that they occur in the same file in which they are used; they are structured in that they have well-defined scope; and they are recursive in that an arbitrary number of extensions may be nested.

Our reflective grammars are based on context-free grammars. Although many modern languages can be made to fit into restricted subsets of context-free languages, such as LALR(1), context-free languages are easier to understand and manipulate, and are closed under composition [12]. This means that they are more suitable for languages which are to be extended by the user.

Others have demonstrated impressive speed improvements to the Earley and GLR algorithms [2, 3, 14–16]. We believe that the historical performance motivations for using restricted subsets of context-free grammars no longer apply.

A macro system could provide meaning to these syntactic extensions, but we do not present one here; this paper only covers parsing.

In section 2, we describe reflective languages in more detail. Section 3 describes a recognition and parsing algorithm. Section 4 proves an upper bound to the time taken by parsing. Sections 5 covers related work, and section 6 discusses our conclusion and future work.

2. REFLECTIVE LANGUAGES

Examples

The crux of our examples is the special right-hand side symbol $\mathbb{R}$. In the grammar G, the strings w that $\mathbb{R}$ derives (denoted $G \vdash \mathbb{R} \Rightarrow w$), are the strings in the set

$$\{w_1 w_2 : G \vdash \langle \text{Gram} \rangle \Rightarrow w_1 \text{ and } G' \vdash S' \Rightarrow w_2\},$$

where

- $\langle \text{Gram} \rangle$ is a distinguished nonterminal in G such that strings derivable from $\langle \text{Gram} \rangle$ can be interpreted as grammars by an operation denoted $[\![-]\!]$.
- $G' = G \oplus [\![w_1]\!]$, where $\oplus$ creates a new grammar by combining the productions of two grammars, and

*This research was made possible by the US National Science Foundation under grant number CCF-0811015, "CPA-SEL: Developing a Theory of Hygienic Macros".

LDTA 2011 Saarbrücken, Germany

- S' is the start symbol of G'.

For our examples, we will define a reflective grammar for a language containing numbers, identifiers, and function invocations in the style of C-like languages. In addition to these conventional elements, the grammar accepts extensions, marked by pairs of curly brackets. The meaning of the extension symbol $\mathbb{R}$ depends on the nonterminal ⟨Gram⟩, which we also must define, giving a BNF-like meta-syntax for reflective grammars. $\mathbb{R}$ is represented in this notation as `REFL`. The start nonterminal of the resulting grammar is specified immediately after `gram`.

We assume that the nonterminals ⟨Identifier⟩, ⟨Nonterm⟩, ⟨QuotedString⟩, and ⟨NaturalNumber⟩ have been given appropriate definitions already. We also assume that whitespace is ignored, except that ⟨Nonterm⟩ and ⟨Identifier⟩ follow standard tokenization rules. Our parser implementation successfully processes all the examples we give.

⟨Expr⟩ → ⟨SimpleExpr⟩`(`⟨Expr⟩⟨MoreArgs⟩`)`
⟨Expr⟩ → ⟨SimpleExpr⟩
⟨SimpleExpr⟩ → ⟨Identifier⟩
⟨SimpleExpr⟩ → ⟨NaturalNumber⟩
⟨SimpleExpr⟩ → `{{ ℝ }}`
⟨MoreArgs⟩ →
⟨MoreArgs⟩ → `,` ⟨Expr⟩⟨MoreArgs⟩
⟨Gram⟩ → `gram <`⟨Nonterm⟩`>` ⟨Prods⟩ `end_gram`
⟨Prods⟩ →
⟨Prods⟩ → ⟨Prod⟩ ⟨Prods⟩
⟨Prod⟩ → `<`⟨Nonterm⟩`> ::=` ⟨RhsItems⟩ `;`
⟨RhsItems⟩ →
⟨RhsItems⟩ → `<`⟨Nonterm⟩`>` ⟨RhsItems⟩
⟨RhsItems⟩ → ⟨QuotedString⟩⟨RhsItems⟩
⟨RhsItems⟩ → `REFL` ⟨RhsItems⟩

A simple sentence in the language of this grammar is `plus(1, plus(2,3))`. A sentence that uses its reflective capabilities to add simple infix operations is

```
plus(1, plus(2,
          {{ gram <Expr>
               <Expr> ::= <SimpleExpr> <Op> <Expr> ;
               <Op> ::= "+" ;
             end_gram
             3 + plus(4, 5 + 6) }} ), 7)
```

The extension recognizes the text between `gram` and `end_gram` inclusive as being derived from ⟨Gram⟩. It interprets the grammar extension, and after that, it expects a string derived from ⟨Expr⟩ in the extended grammar, which it finds: `3 + plus(4, 5 + 6)`. The surrounding text, that is, `plus(1, plus(2, {{ and }}), 7)`, is in the original grammar. This means that the sentence

```
plus(1, plus(2,
          {{ gram <Expr>
               <Expr> ::= <SimpleExpr> <Op> <Expr> ;
               <Op> ::= "+" ;
             end_gram
             3 + plus(4, 5 + 6) }} ), 7 + 8)
```

is *not* in the grammar, because `7 + 8` is outside the $\mathbb{R}$ that provided a new definition for ⟨Expr⟩.

Extensions can be used to gradually build up more powerful languages. In the following example, still in the same base grammar, we add lambda expressions and then infix operations (we represent λ as `\`, making the assumption that backslash is not already used as the escape character in string literals):

```
plus(1,
  {{ gram <Expr>
       <Expr> ::= "\" <Identifier> "." <Expr> ;
       <SimpleExpr> ::= "(" <Expr> ")" ;
     end_gram
     (\x. plus(2,x))(
       plus(3,
         {{ gram <Expr>
              <Expr> ::= <SimpleExpr> <Op> <Expr> ;
              <Op>   ::= "+" ;
            end_gram
            (\y. 4 + y)(
              5 + (\z. 6 + z)(7)) }} )) }} )
```

Note that the extension markers that this base grammar uses, `{{ }}`, have no special status in our system, and the user could choose to use them as another kind of delimiter, provided he or she did so unambiguously. The only reason they appeared in the base grammar at all because omitting them would have made extensions hard to read, and even made it ambiguous where a grammar extension ends after binary operations are permitted.

However, suppose that the author of the base language lacked this foresight, and had written the extension rule as ⟨SimpleExpr⟩ → $\mathbb{R}$, instead of ⟨SimpleExpr⟩ → `{{ ℝ }}`. All would not be lost, because the user could have simply added and then used a new, better construct using `REFL`, which represents the $\mathbb{R}$ construct in our meta-syntax:

```
plus(1, gram <Expr>
          <Expr> ::= "{{" REFL "}}" ;
        end_gram
        {{ gram <Expr>
             <Expr> ::= <SimpleExpr> <Op> <Expr> ;
             <Op> ::= "+" ;
           end_gram
           2 + 3 }} )
```

The old and now ambiguous extension syntax still remains, however. This is because, for simplicity, we have omitted from these examples the ability to remove productions from grammars. It would be very easy to add this, however. Our formalism does not depend on any relationship between the grammar being extended and the extension, but to obtain the complexity bounds of section 4, it must be possible to compute the extension quickly.

Definitions

To define reflective grammars, we first need some metavariables. Let t range over terminal symbols, A and B be nonterminals, α, β, γ, and δ be right-hand sides (strings of terminals, nonterminals, and of the distinguished symbol $\mathbb{R}$), x be the input string of terminals, and let i, j, k, and l be indices into that string. We will use $x_{i,j}$ to represent substrings of x. The indices are zero-based and half-open; i.e., $x = x_{0,|x|}$. The empty string will be represented with the symbol ϵ. We will name other strings w. Finally, we will use G for a reflective grammar.

A reflective grammar G consists of some set of productions $(A \to \alpha) \in G$, and a start symbol $A = G.\text{start}$.

Semantics

In order to define the meaning of a reflective grammar, we must define the meaning of right-hand sides. We write $G \vdash \alpha \Rightarrow x$ to mean that the right-hand side α derives the string x according to the grammar G. Right-hand sides are built recursively from terminals, nonterminals, and the $\mathbb{R}$ symbol:

$$\textsc{L-Empty}\ \frac{}{G \vdash \epsilon \Rightarrow \epsilon}$$

$$\textsc{L-Terminal}\ \frac{G \vdash \alpha \Rightarrow w}{G \vdash \alpha t \Rightarrow w t}$$

$$\textsc{L-Nonterminal}\ \frac{G \vdash \alpha \Rightarrow w_1 \qquad (A \to \delta) \in G \qquad G \vdash \delta \Rightarrow w_2}{G \vdash \alpha A \Rightarrow w_1 w_2}$$

$$\textsc{L-Reflection}\ \frac{\begin{array}{c} G \vdash \alpha \Rightarrow w_1 \qquad G \vdash \langle \text{Gram} \rangle \Rightarrow w_2 \\ G' = G \oplus [\![w_2]\!] \qquad (G'.\text{start} \to \delta) \in G' \qquad G' \vdash \delta \Rightarrow w_3 \end{array}}{G \vdash \alpha \mathbb{R} \Rightarrow w_1 w_2 w_3}$$

We say $x \in L(G)$ (that is, x is in the language of G), iff $G \vdash G.\text{start} \Rightarrow x$.

We restrict $\oplus$ by forbidding the user from extending the special $\langle \text{Gram} \rangle$ nonterminal, and the nonterminals that make it up, because the interpretation function $[\![-]\!]$ is fixed, so it would not be able to interpret the newly-valid strings that $\langle \text{Gram} \rangle$ derives. However, a macro system using this parser could reasonably permit extensions to $\langle \text{Gram} \rangle$ if the user supplied a translation from the extended notation for grammars into the original notation. Also, to make our complexity analysis simpler, we require that $\langle \text{Gram} \rangle$ be non-nullable and appear on the left-hand side of only one production.

3. RECOGNIZER ALGORITHM

We next present an algorithm for recognizing the language of a reflective grammar G, based on the Earley recognizer algorithm [8]:

$$\textsc{R-Start}\ \frac{G.\text{start} \to \delta \ \in G}{(0, G.\text{start} \to \cdot\delta, G) \in S_0}$$

$$\textsc{R-Shift}\ \frac{(i, A \to \alpha \cdot t \beta, G) \in S_j \qquad x_j = t}{(i, A \to \alpha t \cdot \beta, G) \in S_{j+1}}$$

$$\textsc{R-Call}\ \frac{(i, A \to \alpha \cdot B \beta, G) \in S_j \qquad (B \to \delta) \in G}{(j, B \to \cdot\delta, G) \in S_j}$$

$$\textsc{R-Return}\ \frac{(i, A \to \alpha \cdot B \beta, G) \in S_j \qquad (j, B \to \delta \cdot, G) \in S_k}{(i, A \to \alpha B \cdot \beta, G) \in S_k}$$

$$\textsc{R-Parse-grammar}\ \frac{(i, A \to \alpha \cdot \mathbb{R} \beta, G) \in S_j \qquad (\langle \text{Gram} \rangle \to \gamma) \in G}{(j, \langle \text{Gram} \rangle \to \cdot\gamma, G) \in S_j}$$

$$\textsc{R-Refl-call}\ \frac{\begin{array}{c} (i, A \to \alpha \cdot \mathbb{R} \beta, G) \in S_j \qquad (j, \langle \text{Gram} \rangle \to \gamma \cdot, G) \in S_k \\ G' = G \oplus [\![x_{j,k}]\!] \qquad (G'.\text{start} \to \delta) \in G' \end{array}}{\left(k, G'.\text{start} \to \cdot\delta, G'\right) \in S_k}$$

$$\textsc{R-Refl-return}\ \frac{\begin{array}{c} (i, A \to \alpha \cdot \mathbb{R} \beta, G) \in S_j \\ G' = G \oplus [\![x_{j,k}]\!] \qquad \left(k, G'.\text{start} \to \delta \cdot, G'\right) \in S_l \end{array}}{(i, A \to \alpha \mathbb{R} \cdot \beta, G) \in S_l}$$

An Earley recognizer accumulates Earley items. An Earley item is a tuple $(i, A \to \alpha \cdot \beta, G)$, where $(A \to \alpha\beta) \in G$, and the cursor (the $\cdot$ symbol) marks a position in the right-hand side $\alpha\beta$. The grammar G is not part of traditional Earley items; we have added it for our grammars. The algorithm collects sets S_j, where the set S_j corresponds to the jth character in the input string x. The algorithm places the Earley item $(i, A \to \alpha \cdot \beta, G)$ in the set S_j only if $G \vdash \alpha \Rightarrow x_{i,j}$. However, for efficiency's sake, the recognizer only generates that Earley item in the first place if it might be needed (the R-Call rule determines that a nonterminal might need to be recognized at a particular point).

The recognizer proceeds strictly left-to-right. The rules R-Start and R-Call place items of the form $(j, A \to \cdot\delta, G)$ in locations where the nonterminal A is expected to "seed" recognition of an A. The R-Shift rule advances the cursor over an expected terminal. The R-Return rule advances the cursor over an expected nonterminal, provided there exists a corresponding "finished" item of the form $(j, A \to \delta \cdot, G)$.

The last three rules, R-Parse-grammar, R-Refl-call, and R-Refl-return, are our additions to the algorithm. R-Parse-grammar and R-Refl-call are both "seed" rules, analogous to R-call. R-Parse-grammar fires when the recognizer reaches an $\mathbb{R}$, and it starts to consume a string matching $\langle \text{Gram} \rangle$. When the $\langle \text{Gram} \rangle$ has been completely parsed, R-Refl-call creates an extended grammar, and descends into its start terminal. Finally, R-Refl-return is analogous to R-Return; it is triggered by an Earley item that indicates that a string matching the extended grammar is completed, and it advances the cursor over the $\mathbb{R}$ that was waiting on it.

If $G' = G \oplus [\![x_{j,k}]\!]$, then we will say that $G'.\text{location} = (j, k)$ and $G'.\text{parent} = G$ (note that G could be an extended grammar or just the base grammar). We will compare grammars in an intensional fashion. Two extended grammars will be equal exactly when their locations and parents are the same, which implies that, in fact, they posses exactly the same rules. This will decrease the complexity of executing the R-Refl-return rule, and make equality comparisons between Earley items fast.

The algorithm is considered to have recognized the string x in the language G iff it produces an Earley item of the form $(0, G.\text{start} \to \delta \cdot, G)$ in the last set, $S_{|x|}$.

Parsing instead of recognizing

There are two approaches to turn the recognizer into a parser. If ambiguous parses are to be rejected by the parser, Earley's

simple technique suffices: In each Earley item, we associate each nonterminal to the left of the cursor with a pointer to the "completed" Earley item $(j, B \to \delta\cdot, G)$ that derives it. Items that have multiple pointers render any parse that uses them ambiguous.

If a representation of all parses is desired, Scott's Buildtree algorithm [19] can be adapted easily to our recognizer. It depends on the recognizer annotating nodes with "predecessor" and "reduction" pointers. Therefore, when a rule produces an item $(i, A \to \alpha\beta\cdot\gamma, G) \in S_k$, where β is a single terminal, nonterminal, or $\mathbb{R}$, it adds a predecessor pointer from it to the antecedent item $(i, A \to \alpha\cdot\beta\gamma, G) \in S_j$. When R-RETURN produces $(i, A \to \alpha B\cdot\gamma, G) \in S_k$, it adds a reduction pointer from it to the antecendent item $(j, B \to \delta\cdot, G) \in S_k$, and when R-REFL-RETURN produces a rule of the form $(i, A \to \alpha\mathbb{R}\cdot\gamma, G) \in S_k$, it adds a reduction pointer from it to the antecedent item $(j, B \to \delta\cdot, G') \in S_k$.

Scott's algorithm traverses the Earley items and builds up a shared packed parse forest. The symbol nodes [19, p. 59] are marked with a nonterminal and a beginning and ending position. In a reflective setting, these nodes must also have the grammar from which the nonterminal came, because a nonterminal is only meaningful in the context of some grammar.

Correctness

THEOREM 1 (RECOGNIZER CORRECTNESS).

$$G \vdash G.\text{start} \Rightarrow x$$
$$\text{iff there is a production}$$
$$(G.\text{start} \to \gamma) \in G \text{ such that } (0, G.\text{start} \to \gamma\cdot, G) \in S_{|x|}$$

We prove each direction separately. The algorithm is complete:

$$G \vdash \alpha \Rightarrow x_{i,j} \text{ and } (i, A \to \cdot\alpha\beta, G) \in S_i$$
$$\text{implies}$$
$$(i, A \to \alpha\cdot\beta, G) \in S_j$$

by induction on the derivation of $G \vdash \alpha \Rightarrow x_{i,j}$, and the algorithm is sound:

$$(i, A \to \alpha\cdot\beta, G) \in S_j \text{ implies } G \vdash \alpha \Rightarrow x_{i,j}$$

by induction on the derivation of $(i, A \to \alpha\cdot\beta, G) \in S_j$.

The full proof is included in a longer version of this paper [20].

4. COMPLEXITY

We will characterize the complexity of this algorithm in terms of both the length of the input string and the nature of extended grammars it defines. Let n be the length of the input string, and let g be the maximum size of any extended grammar defined. We define the size of a grammar to be the sum of the number of productions and the length of the right-hand sides. By this definition, there are only g distinct values of $A \to \alpha\cdot\beta$ possible in a grammar of size g.

At each input position, there is some set of grammars which might be the current grammar, given the part of the string to the left of the character. Let m be the maximum of the size of these sets, over the length of the string. Having m be greater than 1 occurs in cases where something else shares syntax with a syntax extension construct, or when the extension is not terminated unambiguously, both of which are undesirable in practice. However, in pathological cases, m grows exponentially with n. We know m is always finite because grammar extensions are applied in the order encountered and $\langle\text{Gram}\rangle$ is non-nullable, so every grammar is uniquely defined by sequence of distinct nonoverlapping nonempty substrings of the input string. It is possible to limit the value of m and abort parsing if it exceeds some preset value.

Before we proceed, we must specify the behavior of $[\![x]\!]$ and $\oplus$. We require that both of those take no more than $O(ngm)$ time. Most natural definitions will satisfy this easily, as the string x is no more than n characters long, and the grammars produced by $\oplus$ and $[\![x]\!]$ have size no more than g.

Now we shall prove that recognition takes $O(n^3g^3m^3)$ time. Our argument follows that of Earley [8].

First, we observe that the algorithm can be executed by first determining the contents of S_0, then S_1, and so on, because the contents of each S never depends on an S further to the right. Furthermore, every rule that places an Earley item into set S_i has as an antecedent the existence of an Earley item in S_i, with the exception of R-START and R-SHIFT. Imagining for the moment that each S_i is a set that allows mutation by adding members, we sketch out a strategy for taking the closure of our rules:

For each S_i, in order, "seed" the set by executing R-START if $i = 0$, or R-SHIFT on every appropriate item in S_{i-1} otherwise. Now close the set over the remaining rules: Apply all rules to the new Earley items, the result of which becomes the new Earley items for the next iteration, repeating until no new items appear.

This closure process is the heart of the algorithm. For each Earley item generated, it will execute the rules, and insert the resulting item (if any) into the appropriate set. There is one set of Earley items for each input character, so the asymptotic running time is

$$\text{number-of-input-characters} \times$$
$$\text{number-of-Earley-items-per-set} \times (\text{rule-execution-time} +$$
$$\text{items-produced-per-item} \times \text{set-insertion-time}).$$

There are n input characters. Each set contains at most $O(ngm)$ Earley items: in the form $(i, A \to \alpha\cdot\beta, G_1)$, there are n possible values of i, g possible values for $A \to \alpha\cdot\beta$, and the number of distinct grammars G_1 in the set is limited to m.

If each set is represented as an array of length n containing linked lists of items, and an item anchored at i is stored in the list at index i of the array, there will be at most $O(gm)$ items in each linked list. To perform set insertion by adding elements to these lists, we also need to compare Earley items for equality quickly. It is possible to store all the components of our Earley items as indices for constant-time comparison. This is trivial for the anchor i and for the rule position $A \to \alpha\cdot\beta$, but requires explanation for the grammar G. The contents of grammars can be stored in a table, and each Earley item's reference to the current grammar can be stored as an index into that table. We have required that there only be one production of the form $\langle\text{Gram}\rangle \to \gamma$, so for each grammar with location (i, j) and parent G', there is only one possible Earley item that can produce it via R-REFL-CALL. This means that newly created grammars are unequal to all existing grammars, so the table never needs

to be searched. Therefore, comparing Earley items to each other takes constant time, and therefore inserting an Earley item into the set S_i takes $O(gm)$ time.

Now, all that remains is to determine, per input item, how long the rules take to execute, and how many items the rule produces. Each rule (other than R-START, which takes $O(g)$ time to execute overall) has at least one Earley item as a antecedent. To apply the rule to an Earley item, we substitute the item into the antecedent, and then test the remaining antecedents. This means that rules with two Earley items as antecedents will be attempted twice and succeed the second time.

R-SHIFT This rule takes $O(1)$ time to test the expected terminal against the input string. It produces at most a single item.

R-CALL This rule needs to walk G, so it takes $O(g)$ time, producing at most $O(g)$ items.

R-RETURN We reproduce the rule below:

$$\text{R-RETURN}\quad \frac{(i, A \to \alpha\cdot B\beta, G) \in S_j \qquad (j, B \to \delta\cdot, G) \in S_k}{(i, A \to \alpha B\cdot\beta, G) \in S_k}$$

We will show that the rule takes $O(ngm)$ time and produces $O(ngm)$ items. It is always true that $j \leq k$, because the end of a production must not come before its start. There are two possible ways that an Earley item could be relevant to this rule:[1]

If we have the item $(j, B \to \delta\cdot, G) \in S_k$[2], we know what j is and that all matching items are in S_j. There are $O(ngm)$ items in S_j which need to be checked to see if they match $(i, A \to \alpha\cdot B\beta, G)$. All of them could match: this rule could produce as many as $O(ngm)$ items.

But if we have the item $(i, A \to \alpha\cdot B\beta, G) \in S_j$, the only matching Earley items that could have already been produced are those for which $j = k$. So, we need to search S_j, which takes $O(gm)$ time to produce $O(gm)$ items, because the anchor of the item we are looking for is known to be j. The fact that S_j is only partially complete at this point is of no consequence; whichever item arrives last in S_j will succeed in finding the other.

R-PARSE-GRAMMAR Like R-CALL, this takes $O(g)$ time, producing at most $O(g)$ items.

R-REFL-CALL Computing $G \oplus [\![x_{j,k}]\!]$ takes $O(ngm)$ time, as specified above. $\langle \text{Gram} \rangle$ is required to be non-nullable, so $j < k$, and therefore the $(j, \langle \text{Gram} \rangle \to \gamma\cdot, G) \in S_k$ item always appears last. Searching S_j for items matching $(i, A \to \alpha\cdot\mathbb{R}\beta, G)$ takes $O(ngm)$ time and produces at most $O(ngm)$ items.

R-REFL-RETURN G'.location $= (j, k)$, and G'.parent $= G$. Other than that extra bookkeeping, this rule proceeds like R-RETURN.

For each Earley item, executing the rules takes $O(ngm)$ time and produces up to $O(ngm)$ items. Each item that is produced needs to be inserted into the appropriate set (which, as we saw above, takes $O(gm)$ time). The deduplication performed by set insertion ensures we only have to execute the rules once per *unique* Earley item, even if the item is produced multiple times. Otherwise, execution time would be slower, and it would even diverge in the case of left-recursive rules.

Our total running time therefore is $n \times O(ngm) \times (O(ngm) + O(ngm) \times O(gm)) = O(n^3 g^3 m^3)$. If the rules R-PARSE-GRAMMAR, R-REFL-CALL, and R-REFL-RETURN are omitted, the original Earley algorithm is recovered. The R-RETURN rule, which remains, can still take $O(ngm)$ time and produce $O(ngm)$ items, so the complexity is the same without the reflective rules. Since Earley supports a single grammar of fixed size, g and m are constants. This is consistent with Earley's $O(n^3)$ result. Our system is therefore "pay-as-you-go": its reflective features have no asymptotic cost if they are not used.

Earley recognition provides further performance guarantees in cases where the input obeys certain restrictions. We have not examined whether those same guarantees apply to our work.

Buildtree complexity

The Buildtree algorithm of Scott [19], introduced in section 3, can be used to construct parse trees (based on Earley items) when the results of ambiguous parses are needed in a compact format. (An ambiguous grammar may parse a sentence exponentially many or even infinitely many ways.)

Scott's complexity analysis asserts that Buildtree takes time proportional to

$$\begin{gathered}\text{number-of-input-characters} \times \\ \text{number-of-Earley-items-per-set} \times \\ \text{predecessor-items-per-item}\end{gathered}$$

The number of predecessor items an Earley item may have, as in Scott's work, is n. To see this, observe that an item where the cursor follows a nonterminal,

$$(i, A \to \alpha B\cdot\beta, G) \in S_j$$

can have as predecessor any item of the form

$$(i, A \to \alpha\cdot B\beta, G) \in S_k$$

where $0 \leq k \leq j$. This same argument applies to cases where the cursor follows a $\mathbb{R}$.

On the other hand, if the cursor follows a terminal, there is exactly one predecessor, and items where the cursor is at the beginning of the right-hand side have no predecessor.

The number of input characters is n. As above, the number of Earley items in each of our sets is $O(ngm)$. So executing Buildtree requires $O(n^3 gm)$. This means that Buildtree,

[1] Here, we differ from Earley by omitting a small optimization; he only tests items for applicability as the $(j, B \to \delta\cdot, G)$ antecedent in the R-RETURN rule. This always works when $j < k$, and sometimes works when $j = k$. Additional work must be done to make this behave correctly in the presence of nullable productions. Aycock [2] discusses three different solutions to this problem.

[2] An anonymous reviewer points out that the value of δ is irrelevant in executing this rule. therefore, an intermediate rule could collapse all items of the form $(i, B \to \delta\cdot, G) \in S_k$ into a special item $(i, B \to \square, G) \in S_k$, which the R-RETURN rule could look for instead, reducing the number of times it executes. However, this would not have an asymptotic effect on performance; the number of distinct possible values of $B \to \square$, like the number of distinct possible values of $B \to \delta\cdot$, is in $O(g)$.

which takes place only once (after recognizing is completed), requires less time than recognizing, so it does not affect the overall complexity.

5. RELATED WORK

Parsers

The idea of modifying an Earley parser to parse a more powerful class of grammars was inspired by YAKKER [11], a powerful Earley-based parser for dependent grammars. A dependent grammar can, for example, recongize the language of strings containing a literal number n followed by a sequence of precisely n characters.

Derivative-based parsing [17] is an approach to parsing context-free languages in which the parse state at a given character is simply a grammar representing the language of strings that are valid suffixes to the already-parsed portion. The authors suggest that it could be used to implement reflective grammars, but supply no details.

Like context-free grammars, parsing expression grammars (PEGs) can be composed by combining productions to produce a legal grammar [10]. However, the ordered choice provided by PEGs is not a true union, and "incorrect orderings can cause suble errors" [12]. For example, adding an `if...then` construct can turn an existing `if...then...else` construct into a syntax error.

Language extension systems

There are a variety of systems that tackle the issue of syntax extensibility. Each work in this category is a complete system that tackles both the issue of parsing and the issue of transformation. We will only cover the comparable portion here, the parsers.

A few of these systems parse input using some kind of dynamic grammars which, like ours, support multiple grammars in one file.

Kolbly [13] describes a syntax extension system with an Earley-based parser that can parse different regions of a file in different grammars. However, all grammar extensions must be predefined by the language designer — the user cannot extend the language.

Another macro system with flexible syntax is ZL [1]. It allows new syntax to be added to C, though a system of iterated re-parsing. However, it restricts what syntactic forms the user may add.

Although Dylan's macro system [4] does not involve any special parser technology, it does loosen Lisp's parentheses to a "syntactic skeleton", giving macro authors more control over the appearance of macro invocations.

Gel [9] is a language syntax that, by requiring adherence to whitespace conventions, correctly parses code that looks like Java, CSS, Smalltalk, and ANTLR. Their goal is in some ways a mirror image of ours: they unify a set of existing syntaxes into one large syntax, while we describe how a single small syntax can be extended into many others in the bounds of one file.

The Silver project [21] is a system for describing and extending languages, and transforming those languages using attribute grammars. Schwerdferger and Van Wyk describe [18] a static analysis for language extensions which ensures that, given a host language, any number of these extensions can be added to the host language, and the result will be LALR(1), as their parser requires. However, they must significantly restrict the permissible forms of syntax extensions in order to do so.

Metafront [5] is a system for defining languages and transformations between them. They describe a novel type of grammar called a "specificity grammar". In such a grammar, more specific productions have priority over less specific productions. Although composing their grammars can produce errors, these errors can be expressed entirely in terms of the productions involved, rather than as confusing shift/reduce and reduce/reduce conflicts. They also have what they describe as a macro system; however, their macro definitions always have the scope of an entire file, so they can use existing parser technology.

A system described by Cardelli, Matthes, and Abadi [6] discusses incrementally extending grammars by adding productions (and grammar restriction, where productions are removed). It rejects compositions of grammars that are not LL(1), but provides powerful integration between grammar definitions and transformations.

Camlp4 [7] is a preprocessor for the Ocaml language. It allows the user to extend the Ocaml syntax. It allows the language designer to select what parser the resulting, extended, language will be parsed with, but the user must select one language per file.

6. CONCLUSION AND FUTURE WORK

We have defined a class of grammars that specify languages that can modify their own syntax during parsing. We have presented an algorithm that can parse these reflective grammars and can parse nonreflective grammars as fast as an ordinary Earley parser. Furthermore, we have placed bounds on how costly the reflective feature is, in terms of how it is used.

We intend this work as the first step in building a macro system applicable to languages that lack parenthesis-based syntax. Our next steps will be to define requirements for a powerful and usable macro system, and describe how such a macro system would interact with this parser. In such a system, there would be no special syntax for macro invocation, so user-defined syntax would be indistinguishable from core syntax. With the dynamic power of our parser, it would be possible to have local definitions for macros, and even to import macros in a restricted scope.

7. REFERENCES

[1] K. Atkinson, M. Flatt, and G. Lindstrom. ABI compatibility through a customizable language. *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering - GPCE '10*, page 147, 2010.

[2] J. Aycock. Practical Earley parsing. *The Computer Journal*, 45(6):620–630, June 2002.

[3] J. Aycock and N. Horspool. Directly-executable Earley parsing. In R. Wilhelm, editor, *Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 229–243. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45306-7_16.

[4] J. Bachrach and K. Playford. D-expressions: Lisp power, Dylan style. `http://people.csail.mit.edu/jrb/Projects/dexprs.htm`, 1999.

[5] C. Brabrand, M. I. Schwartzbach, and M. Vanggaard. The metafront system: Extensible parsing and

transformation. *Electronic Notes in Theoretical Computer Science*, 82(3):592–611, Dec. 2003.

[6] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. `http://lucacardelli.name/Papers/SRC-121.ps`, 1994.

[7] D. de Rauglaudre. Camlp4 - reference manual. `http://caml.inria.fr/pub/docs/manual-camlp4/`, 2003.

[8] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 26(1), 1970.

[9] J. Falcon and W. Cook. Gel: A generic extensible language. In *Domain-Specific Languages*, pages 58–77. Springer, 2009.

[10] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings ACM Symposium on Principles of Programming Languages*, pages 111–122, 2004.

[11] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. *Annual Symposium on Principles of Programming Languages*, 45(1), 2010.

[12] L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. *Proceedings of Onward! 2010*, 2010.

[13] D. M. Kolbly. *Extensible Language Implementation*. Ph.D., University of Texas at Austin, 2002.

[14] Y. Mandelbaum and T. Jim. Efficient Earley parsing with regular right-hand sides. *Workshop on Language Descriptions Tools and Applications*, 2009.

[15] P. McLean and R. Horspool. A faster Earley parser. In T. Gyimóthy, editor, *Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 281–293. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61053-7_68.

[16] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. *Compiler Construction*, 2004.

[17] M. Might and D. Darais. Yacc is dead. `http://arxiv.org/abs/1010.5023`, Oct. 2010.

[18] A. C. Schwerdfeger and E. R. Van Wyk. Verifiable composition of deterministic grammars. *Conference on Programming Language Design and Implementation*, 44(6), 2009.

[19] E. Scott. SPPF-style parsing from Earley recognisers. *Electron. Notes Theor. Comput. Sci.*, 203:53–67, April 2008.

[20] P. Stansifer and M. Wand. Parsing reflective grammars. Technical report, Northeastern University, 2011. `http://arxiv.org/abs/1102.2003`.

[21] E. R. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Electronic Notes in Theoretical Computer Science*, 203(2):103–116, Apr. 2008.