Just-in-time Compilation for Generalized Parsing

Stefan Fehrenbach

Fachbereich Mathematik und Informatik
Philipps-Universität Marburg

A thesis submitted for the degree of
Master of Science

Supervisors:
Tillmann Rendel
Paolo G. Giarrusso
Prof. Dr. Klaus Ostermann

1 September 2014

Abstract

Parsing syntactically extensible languages requires generalized parsers which are slow to generate for repeatedly changing grammars. This situation is similar to the execution of dynamic languages like JavaScript, suggesting that we can appropriate technology from that field to use in just-in-time compiled parsers. We implement two just-in-time compiling grammar interpreters, a simple one and a generalized LL parser, using a research prototype of a programming language implementation framework called Truffle. Exploratory performance experiments suggests that appropriating just-in-time compilation technology for parsers is not only possible, but close to being competitive for generalized parsing and might even surpass handwritten recognizers in some cases.

Contents

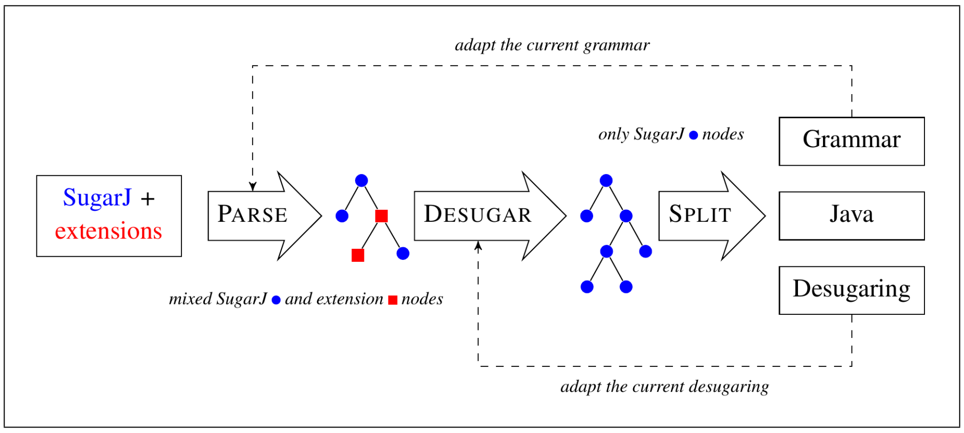
```
package demo;

import SQL;
import XML;

class Demo {
  public String getItemXML(int itemID) {
    Query q = em.SELECT name, desc
                  FROM Item i
                  WHERE i.id == ${itemID};
    return <item>
                <id>${itemID}</id>
                <name>${q.getResult().get("name")}</name>
                <description>${q.getResult().get("desc")}</description>
           </item>;
  }
}
```

Figure 1: Database query and XML serialization with SugarJ.

1 Introduction

Syntactically extensible programming languages prove useful for implementing domain-specific languages. Where pure embedding and similar techniques allow for expressing domain-specific semantics in a host language, the flexibility to almost arbitrarily extend the language syntax allows programmers to use even domain-specific notation. One example of such a language is SugarJ [4]. It is based on Java, but it can be extended through sugar libraries, which extend the base language with new syntax, semantics, static analysis, and IDE support [3]. See Figure 1 for an example program in SugarJ. The getItemXML method fetches an item from a database and serializes said item to an XML string. The program imports and uses two domain-specific languages: one for database queries and another for literal XML syntax.

From personal experience [5] we can confirm that working with SugarJ is quite pleasant, except for one thing: waiting for parser generation. Parsing SugarJ files is a challenge because the language may change while a file is being parsed. Consider the program in Figure 1 again. The first line is plain Java and declares the file to belong to the package demo. The first import statement reads **import** SQL;. In Java this would just bring a number of classes into scope. In SugarJ this import statement changes the language. After the import, we can suddenly write database queries as SQL code in the middle of the Java code. The next import extends the language further with XML literals. The syntax of language extensions may even interact



adapt the current grammar
only SugarJ ● nodes
Grammar
SugarJ + extensions
PARSE
DESUGAR
SPLIT
Java
Desugaring
mixed SugarJ ● and extension ■ nodes
adapt the current desugaring

Figure 2: SugarJ processing loop. Illustration by Sebastian Erdweg [4].

with existing Java syntax and that of other extensions, as we see in the XML part, where we have some inline Java code again.

Currently, SugarJ implements parsing the changing language as illustrated in Figure 2. We start parsing with the SugarJ base grammar, which is the Java grammar plus the languages that are used to implement language extensions, namely SDF for describing grammars and Stratego for implementing the semantics. Using the base grammar, we parse one top-level declaration at a time. In our example from Figure 1 that would be the package declaration. A package declaration does not change the language, so we continue to parse the rest of the file, starting with the next top-level declaration. In our example, the next top-level declaration imports the SQL language extension. Thus we need to adapt the current grammar, which is the SugarJ base grammar, to include the productions that allow the programmer to use SQL in the rest of the file. SugarJ uses SDF to describe the (mostly) context-free languages and language extensions. Thus it is possible to just combine the current grammar with the extension grammar to get a grammar to be used for the rest of the file. Now SugarJ generates a parser for the new grammar by invoking the Scannerless GLR parser generator that is part of the SDF reference implementation. This takes a while. We continue parsing the next top-level declaration, which again imports a language, so we have to change the grammar again and generate a new parsers for the combined language.

SugarJ caches parsers, so if we use the same combination of languages again, it will not regenerate the same parsers over and over again. When developing language extensions though, programmers frequently change the grammar of their language. To see the effect of grammar changes, a programmer most likely has one or more files opened in the SugarJ IDE that exercise the language extension. Depending on the order of inputs, SugarJ will need to generate many parsers to parse, compile, and display the test files, which will take minutes. We think we can do better.

Let's consider the setting again. We have a grammar with which to parse a file and expect the process to be responsive, not take minutes for parser generation. A web browser faces a similar scenario when presented with a website that includes JavaScript code. To display the website, it must execute the JavaScript code and because users do not like to wait for programs, the response time has to be as low as possible. Nowadays however, JavaScript is also used for more extensive computation, so peak performance is a concern. The solution is just-in-time compilation. Browser start executing JavaScript code as quickly as possible by interpreting code with minimal or no optimizations at first. While the code runs in the interpreter, the browser collects information about the executed code, for example which branches are taken and how often. This information is used to optimize and compile frequently executed code to fast machine code in the background. Similar considerations apply to parsing. Thus, the question that inspired this thesis: what would happen if we use *just-in-time compilation for generalized parsing?*

- SDF generates a parse table, which is an optimization that trades ahead-of-time compilation costs for the faster parse times of a table-based parser. Could we instead just start parsing with the grammar as it is?
- Many files will not exercise the entire grammar. For example, only files that define language extensions require the entirety of the SDF and Stratego grammars. Similarly, sugar libraries frequently contain little to no Java code. Could we avoid costly optimizations for parts of the grammar that are not used (often)?
- Importing an additional language extensions generally only changes a small amount of nonterminals in the existing grammars. The existing language stays mostly the same; the additional language is mostly self-contained; there are only a small set of interaction points between languages. Can we incrementally change the parser, retaining the parts that are not affected by the changes?
- Many of smart people work on just-in-time compilers for programming languages. Can we apply their work to parsing?

Truffle [14, 15] is a recent research project that tries to make it easy to write high-performance virtual machines for programming languages. Write an AST interpreter in a certain style and Truffle will use partial evaluation to specialize the interpreter to the program, making it a just-in-time compiler. A parser is the first Futamura projection of a grammar interpreter [6, 9]. Thus it looks like all we need to do is write an interpreter for context-free grammars.

SugarJ allows combination of arbitrary context-free languages. Therefore we need a parsing algorithm that supports any context-free language, like Earley, generalized LL, or generalized LR. We chose to use a generalized LL parser (GLL), because we had access to an implementation in Java and Truffle needs an AST interpreter written in Java. Truffle also enforces a particular style and requires users to be very explicit about static data, which in our case is the grammar, and control flow which turns out to be very difficult for GLL.

In summary, this thesis makes the following contributions:

- Section 2 gives a short introduction to parsing by discussing how to systematically generate recursive descent recognizers from a grammar. This is mostly background information for readers who are unfamiliar with writing a simple parser by hand. We later refer to these handwritten recognizers in a performance comparison with Truffle-enabled grammar interpreters.

- In Section 3 we discuss how a grammar interpreter can perform the same job. The implementation itself is straightforward and only becomes a major contribution in Section 4. There we discuss how we modified the grammar interpreter so that it works with Truffle. We thereby demonstrate that Truffle can be used to perform just-in-time compilation of something that is not, not obviously at least, a programming language.

- In the same section, we also describe the first optimization we implemented. Since nonterminals rarely change, we perform inline caching, complete with guards and deoptimization in case they do change.

- In an exploratory performance study (Section 6), we compare the Truffle-based interpreters with and without optimization with handwritten recognizers. The optimized Truffle-based interpreters outperform the handwritten ones, which suggests that this line of work deserves further attention.

- We convert the code of an existing GLL parser to use Truffle. This is similar to what we did for the simple recursive descent grammar interpreter but due to the nature of the GLL algorithm, this proves more a lot more challenging. We discuss the implementation and its problems in Section 5.

- In a second exploratory study (Section 7), we compare the original GLL parser to the Truffle-enabled one. The Truffle-enabled parser is only about 20% slower. This is not unreasonable and suggests that addressing the problems with the implementation and further research into incremental grammar analysis might be worthwhile.

⟨*S*⟩ ::= ⟨*SExp*⟩ `EOF`

⟨*SExp*⟩ ::= ⟨*Atom*⟩ | ⟨*Pair*⟩

⟨*Atom*⟩ ::= ⟨*Symbol*⟩ | ⟨*Number*⟩

⟨*Pair*⟩ ::= ‘`(`’ ⟨*SExp*⟩ ‘`.`’ ⟨*SExp*⟩ ‘`)`’

⟨*Symbol*⟩ ::= `[a-zA-Z]+`

⟨*Number*⟩ ::= `[0-9]+`

Figure 3: Grammar of a simple s-expression language.

2 Background: Recursive Descent Recognizers

In this section, we will see how to systematically turn a grammar into a program that determines whether its input is in the language the grammar describes or not. We say such a program solves the word problem and we call it a recognizer. We use the simplest algorithm possible, a recursive descent recognizer without lookahead. This imposes several restrictions on the grammars and therefore the class of languages we can recognize with such a parser is quite limited. There can be no left recursion, or the recognizer would run into an infinite loop. No lookahead means as soon as we parse a character we are committed. There is no backtracking after parsing a character. We can only backtrack when the current character is not what we expected. There are not many more practical languages with such a grammar but for illustration purposes the simplicity is ideal.

S-expressions are simple enough to keep to the restrictions of our simple recursive descent algorithm and will serve as an example, see the grammar in Figure 3. The start symbol is called ⟨*S*⟩, and its right-hand side says that we expect exactly one s-expression as parsed by ⟨*SExp*⟩ and then the end of the string, as indicated by the special nonterminal symbol `EOF`. An ⟨*SExp*⟩ is either an ⟨*Atom*⟩ or a ⟨*Pair*⟩. A pair is in dotted pair notation, where both constituents are s-expressions themselves, they are separated by a dot and surrounded by parentheses. Atoms are either numbers or symbols. We use regular expression-like syntax for brevity, but we could express the character classes numerals and alphabetic characters using only single literals and additional nonterminals.

We have an informal idea of what the semantics should be. Let’s refine that by translating the s-expression grammar into Java code (Figure 4). Most of everything in Java starts with a class, we define one called RecursiveDescent.

```java
class RecursiveDescent {
   char[] string;
   int pos;

   RecursiveDescent(String string) {
      this.string = string.toCharArray();
      this.pos = 0;
   }

   boolean s() { return sexp() && eof(); }

   boolean sexp() { return atom() || pair(); }

   boolean atom() { return symbol() || number(); }

   boolean pair() {
      return character('(') && sexp() && character('.') && sexp() && character(')'); }

   boolean symbol() { return regex("[a-zA-Z]+"); }

   boolean number() { return regex("[0-9]+"); }

   boolean character(char c) {
      if (string[pos] == c) {
         pos++;
         return true;
      }
      return false;
   }

   boolean eof() { return pos == string.length; }

   // Call with e.g. "(x.(y.(z.NIL)))"
   public static void main(String[] args) {
      System.out.println(new RecursiveDescent(args[0]).s());
   }
}
```

Figure 4: Recognizer for the simple s-expression language.

There are two pieces of state that we need to keep track of: the string we are parsing and the position we are at currently. We represent them as fields and initialize them in a constructor.

Parsing starts with the start symbol $\langle S\rangle$. There are two possible results, the string is in the language, or not. Thus we create methods for nonterminals with the return type **boolean**. The first one is the method **boolean** `s()`. In the grammar, we see that $\langle S\rangle$ has only one alternative: parse the string according to $\langle SExp\rangle$ and then be at the end (`EOF`). Thus, the body of `s()` returns the result of parsing with $\langle SExp\rangle$, that is, it calls the method `sexp()`, and also makes sure that we reached the end of the string by calling `eof()` and combining the results with logical AND. The method `sexp()` is the translation of the nonterminal $\langle SExp\rangle$. If we look again at the grammar, we see that $\langle SExp\rangle$ is an alternative. An $\langle SExp\rangle$ is either an atom as parsed by $\langle Atom\rangle$ or a pair as parsed by $\langle Pair\rangle$. Translated to Java this means we try parsing according to $\langle Atom\rangle$ by calling `atom()`, if and only if that fails, we also try to parse according to $\langle Pair\rangle$ by calling `pair()`. If either of the two succeeds we return **true**, and **false** otherwise. We translate the remaining nonterminals analogously.

In the translation of $\langle Pair\rangle$ we call the method `character` to parse a single character. Its implementation is straightforward. If the character at the current position is the one we expect here, we advance the current position and signal success by returning **true**. Otherwise, we do not advance the current position and return **false**. This is where the somewhat limited backtracking with no lookahead comes from.

The method `eof()` is the translation of the somewhat special terminal symbol `EOF`. It signals that we expect to have reached the end of the string. This is the case iff the current position is the same as the string's length. We omit the implementation of the `regex` method. As mentioned before, this is merely syntactic sugar for repeatedly parsing one of a long list of alternative characters. We can, of course, run the parser by initializing the state and calling the method that matches the start symbol.

3 Background: Interpreting Grammars

In the previous section, we saw how to systematically construct a recognizer for a given language by translating a grammar into Java code. One could of course write a program that performs the translation. Such a program would be called a parser generator. If we consider the grammar to be a programming language, a parser generator is a compiler. It turns the input language program into a program in the output language, in our case Java. A popular alternative to writing compilers is writing interpreters. In this section we will describe how to write an interpreter that takes a grammar and a string as inputs and tells us whether the string is in the language

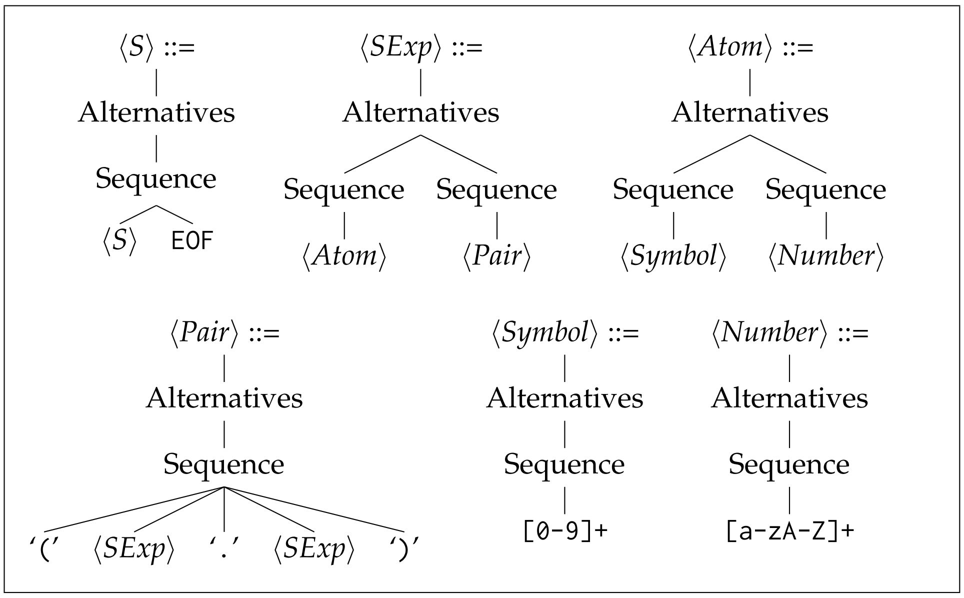


Figure 5: Grammar AST forest for the simple s-expression language.

defined by the grammar. It performs the same job as the recognizer from the last section but, additionally, is flexible in the grammar it considers.

The simplest interpreters are structurally recursive. We want to construct a simple interpreter, so lets consider the structure of our language of grammars. There is the rule definition operator ::=. This suggests we need an environment in which to look up names that are defined by nonterminals that appear on the left hand side of a definition, when we need what they stand for when they appear on the right hand side. The right hand side of a rule is a list of alternatives. Every alternative consists of a sequence of leave nodes in our abstract syntax tree of grammars. The leaf nodes are parsing a single literal symbol and parsing according to a nonterminal symbol. There are two special kinds of nodes, one that recognizes the end of the string and one that deals with our regular expression-like shorthand for parsing multiple characters of a particular class. See the forest of abstract syntax trees in Figure 5, that corresponds to the s-expression grammar from Figure 3.

The common super class of our grammar interpreter's AST nodes will be `GrammarNode`, see Figure 6. Every AST node will have access to some shared state encapsulated by the `ParserState`. The shared state will be the string that is to be parsed, the current position inside the string, and the environment that maps nonterminals to their right hand sides. We also declare the method **boolean** `executeParse()`. In usual programming language interpreters we would probably call it `eval`.

```
abstract class GrammarNode {
   final ParserState state;
   GrammarNode(ParserState state) { this.state = state; }
   abstract boolean executeParse();
}
```

Figure 6: Common super class of our grammar interpreter AST nodes.

We implement the four grammar node classes as detailed in Figure 7. The right hand side of a definition is a list of alternatives. The Alternatives node is the corresponding node in our grammar AST. The alternatives inside an Alternatives node are an array of Sequence nodes. We omit the constructor that initializes this array. To parse a list of alternatives we try to parse according to the first child node, by calling its executeParse() method. If that succeeds, we return true. Otherwise, we try the next until there are none left, in which case we signal failure by returning **false**. Similarly, we parse a Sequence node by recursively calling executeParse() on its children. The difference, is that all of them must succeed for the sequence to succeed. In other words, we fold logical OR and logical AND, respectively, over an array of child nodes.

The leaf nodes not any more complicated. To parse a terminal symbol, we get the character at the current position in the input string out of the parsing state. We succeed and advance one position in the input string if it is the same as the code point c we expect and fail (and do not advance), otherwise.

To parse a nonterminal, we look up its name in the environment that is stored in the parsing state. The result of the lookup is the right hand side of a definition, an Alternatives node. We just call its executeParse() method and return the result. Notice that we perform repeated lookups every time we attempt to parse a nonterminal. On the one hand, this allows us to change the grammar easily. Consider the SugarJ use case. The programmer imports an additional language extension. We build a grammar AST for the added language and union the environments to get the environment for the combined language. In case there are nonterminals that appear in both environments, we would need to union the alternatives for the respective nonterminals. Note that we only perform work for the new language and the part of the current language that overlaps with it. On the other hand, this probably incurs a performance penalty compared to fixing the right hand side of a nonterminal in place. In the next section we will make the assumption that grammars rarely change and optimize for that case by inlining the right hand side of a nonterminal, thus avoiding the lookup, having our cake, and eating it too.

```
class Alternatives extends GrammarNode {
  private final Sequence[] alternatives;

  boolean executeParse() {
    for (Sequence alternative : alternatives)
      if (alternative.executeParse())
        return true;
      else
        continue;
    return false;
  }
}

class Sequence extends GrammarNode {
  private final GrammarNode[] sequence;

  boolean executeParse() {
    for (GrammarNode grammarNode : sequence)
      if (grammarNode.executeParse())
        continue;
      else
        return false;
    return true;
  }
}

class TerminalSymbol extends GrammarNode {
  private final int c;

  boolean executeParse() {
    int currentCodePoint = state.getCurrentChar();
    if (currentCodePoint == c) {
      state.incCurrentChar();
      return true;
    }
    return false;
  }
}

class NonterminalSymbol extends GrammarNode {
  private final String nonterminalName;

  boolean executeParse() {
    Alternatives nonterminalRightHandSide = state.lookupInEnv(nonterminalName);
    return nonterminalRightHandSide.executeParse();
  }
}
```

Figure 7: Grammar interpreter nodes. (Constructors elided.)

```
abstract class GrammarNode extends Node {
   final ParserState state;
   GrammarNode(ParserState state) { this.state = state; }
   abstract boolean executeParse( VirtualFrame frame );
}
```

Figure 8: Common superclass with Truffle. Compare to Figure 6.

4 Just-in-time Compilation of a Grammar Interpreter with Truffle

Interpreters are slow. Truffle turns interpreters into fast [14] just-in-time compilers. And the best part is, it only requires moderate effort on the side of the language implementers. Truffle just-in-time compilers start as AST interpreters, just like the one we saw in the previous section.

We start by making our AST base class extend the Node class provided by Truffle. This base class for nodes provides a couple of helper functions that are needed for node rewriting, which is core to performance optimizations with Truffle. Figure 8 shows the new Truffle-enabled base class for our AST nodes, Truffle-specific changes are highlighted. The second change is that our executeParse method takes a parameter of type VirtualFrame. Frames are Truffles representation stack frames. When calling functions using Truffle you have to pass parameters using frames. Local variables can also be stored there. Since our interpreter did not pass any parameters and there are no locals, we will never need this argument. We still need to declare it because Truffle uses it as a heuristic for which method calls to partially evaluate.

Three of the AST node classes from the old interpreter are easily converted to use Truffle, see Figure 9. Of course, the executeParse method has to accept its virtual frame parameter. When we recursively call executeParse, we just pass the frame we got ourselves.

Truffle uses online partial evaluation, but we still need to annotate what should be considered static data, not unlike the two-level languages commonly used for offline partial evaluation. Static data is in our case the grammar, the program, as represented by the AST. The @Children annotations are there to tell Truffle that those fields hold references to child nodes in the AST. Truffle regards the AST as static, that is, it will never change, except when using the AST rewriting methods from the base Node class. Internally, Truffle uses this invariant to partially evaluate calls into the AST, like when we call executeParse on one of the alternatives in Alternatives node. It will aggressively inline and generate fast machine code instead of performing actual Java method calls.

```
class Alternatives extends GrammarNode {
   private final @Children Sequence[] alternatives;

   @ExplodeLoop
   boolean executeParse( VirtualFrame frame ) {
      for (Sequence alternative : alternatives)
         if (alternative.executeParse( frame ))
            return true;
         else
            continue;
      return false;
   }
}

class Sequence extends GrammarNode {
   private final @Children GrammarNode[] sequence;

   @ExplodeLoop
   boolean executeParse( VirtualFrame frame ) {
      for (GrammarNode grammarNode : sequence)
         if (grammarNode.executeParse( frame ))
            continue;
         else
            return false;
      return true;
   }
}

class TerminalSymbol extends GrammarNode {
   private final int c;

   boolean executeParse( VirtualFrame frame ) {
      int currentCodePoint = state.getCurrentChar();
      if (currentCodePoint == c) {
         state.incCurrentChar();
         return true;
      }
      return false;
   }
}
```

Figure 9: Truffle-enabled nodes. Compare to Figure 7.

```
class UnoptimizedNonterminalCall extends GrammarNode {
  private final String nonterminalName;
  @Child IndirectCallNode indirectCall =
        Truffle.getRuntime().createIndirectCallNode();

  boolean executeParse(VirtualFrame frame) {
    CallTarget nonterminalRightHandSide = state.lookupInEnv(nonterminalName);
    return (Boolean) indirectCall.call(frame,
      nonterminalRightHandSide , new Object[0]) ;
  }
}
```

Figure 10: Simple nonterminal calls. Compare to Figure 9.

The @ExplodeLoop annotation tells Truffle that the loop is over static data and should be unfolded during partial evaluation. It is our humble opinion that Truffle should be able to figure that out by itself, since we already annotated the fields that hold children, but it does not seem to do so.

Truffle ASTs must be trees. Being a directed, even acyclic, graph is not good enough. Cycles are bad because they make Truffle silently stop working. Nodes with more than one parent node are even worse. They seem to work just fine but result in unspecified behavior. Grammars can have cycles and nonterminals can occur at more than one position on a right hand side. Consider our simple s-expression language from Figure 3. There, $\langle Pair \rangle$ appears on the right hand side of $\langle SExp \rangle$ and $\langle SExp \rangle$ appears on the right hand side of $\langle Pair \rangle$, forming a cycle. The way to break cycles is to not put the nodes as children directly, but make the possibly cyclic reference known to Truffle by using its support for functions. Thus, to break the cycles in our grammars, we treat nonterminal definitions like function definitions and appearances of nonterminals on the right hand side as calls to the respective function.

We start with a simple version that uses Truffles special support for defining and calling functions to replicate the behavior of our interpreter from the previous section. The UnoptimizedNonterminalCall node in Figure 10 is the Truffle-enabled version of the NonterminalSymbol node from Figure 7. We declare one child node that is provided by Truffle, the IndirectCallNode. It can be used to make function calls by calling its call method. It is called an indirect call, because we pass the function object as an argument during runtime. This function object is the CallTarget. We also changed the parsing state to cache call targets, not AST nodes directly. Call targets basically wrap a part of the AST in a parent-less root node, that knows how to execute itself.

So in essence, we do the same thing as before: we look up the right hand side of the nonterminal we want to parse in the environment and parse according to it.

Time for the first optimization. This is where Truffle really shines. Most of the time, a nonterminal will not have changed from what it was the last time we looked it up and called it. We hope to make parsing faster by caching the lookup. In traditional just-in-time compiler lingo we would like to have a (monomorphic) inline cache for the result of the lookup.

We introduce two new AST node types: an UninitializedNonterminalCall and a CachedNonterminalCall. An uninitialized nonterminal call is the state of a nonterminal on the right hand side that has not yet been invoked. See the implementation in Figure 11. Upon execution for the first time, this node will look up the right hand side of the nonterminal in the environment and replace itself with a CachedNonterminalCall node that caches the result of the lookup. There is an alternative mode of operation where it replaces itself with an UnoptimizedNonterminalCall like we saw before. We use this to compare the two alternatives against each other. To avoid runtime overhead, we tell Truffle that the kind of replacement we want to perform will never actually change at runtime using the CompilationFinal compiler directive. This allows Truffle to partially evaluate the **switch** statement and only keep the appropriate branch. This might be unnecessary, because we also assert that the UninitializedNonterminalCall will never actually be part of compilation. During the first execution, the node will replace itself and Truffle will only compile nodes that have been executed more often than some configurable threshold.

Similar to the UnoptimizedNonterminalCall, the CachedNonterminalCall uses a special kind of child node to actually call the function. In this case this is a DirectCallNode, as seen in Figure 12. The difference to an indirect call is that we can not change the call target at every invocation of the call method, instead we pass the CallTarget (Truffle's representation of a function body) during initialization. This is where the caching happens. The CallTarget is the result of the lookup we did when we replaced the UninitializedNonterminalCall node by this node. We cache the function body inside the DirectCallNode.

When we execute the CachedNonterminalCall we want to call the cached function body. Before we do, we need to check that our assumption holds, namely that this nonterminal has not been changed since the last time we called this nonterminal. Truffle provides specialized support for this. Next to the function body, we store an Assumption object. An assumption can be checked and invalidated. Before we call the function body that is cached inside the DirectCallNode, we call the Assumption's check method. If the assumption does not hold any longer, that is, has been explicitly invalidated, it will throw an InvalidAssumptionException. Should that happen, we revert to the uninitialized state, where we will do a new lookup and again cache the result.

```
class UninitializedNonterminalCall extends GrammarNode {
   enum CallNodeType { UNOPTIMIZED, CACHED }

   @CompilerDirectives.CompilationFinal
   static CallNodeType callNodeType = CallNodeType.CACHED;

   final String nonterminalName;

   boolean executeParse(VirtualFrame frame) {
      CompilerAsserts.neverPartOfCompilation();

      GrammarNode replacementNode;

      switch (callNodeType) {
         case UNOPTIMIZED:
            replacementNode =
                  new UnoptimizedNonterminalCall(state, nonterminalName);
            break;
         case CACHED:
            CallTarget alternatives = state.lookupInEnv(nonterminalName);
            replacementNode =
                  new CachedNonterminalCall(state, alternatives, nonterminalName);
            break;
      }

      replace(replacementNode, "Call nonterminal " + nonterminalName);

      return replacementNode.executeParse(frame);
   }
}
```

Figure 11: Uninitialized call site of a nonterminal.

```
class CachedNonterminalCall extends GrammarNode {
   final String nonterminalName;
   final Assumption grammarUnchanged;

   @Child private DirectCallNode directCallNode;

   CachedNonterminalCall(ParserState p, CallTarget alternatives,
                                       String nonterminalName) {
      super(p);
      this.nonterminalName = nonterminalName;
      this.grammarUnchanged = p.cacheNonterminal(nonterminalName);
      directCallNode = Truffle.getRuntime().createDirectCallNode(alternatives);
   }

   boolean executeParse(VirtualFrame frame) {
      try {
         grammarUnchanged.check();
         return (Boolean) directCallNode.call(frame,  new Object[0]);
      } catch (InvalidAssumptionException e) {
         GrammarNode replacementNode =
               new UninitializedNonterminalCall(state, nonterminalName);
         replace(replacementNode, "Nonterminal " + nonterminalName + " changed");
         return replacementNode.executeParse(frame);
      }
   }
}
```

Figure 12: Call of a nonterminal with inline cache.

According to Phil Karlton “there are only two hard things in Computer Science: cache invalidation and naming things.” Next to the environment that stores nonterminal function bodies, we store a reference to the same `Assumption` object that guards the inline cache. When we change the right hand side of a nonterminal we invalidate the matching assumption. It might be more efficient to have a single assumption for the whole grammar. This choice is open for debate. One would need a comprehensive, real world benchmark suite for changing grammars for a definitive answer. As it is, the behavior mimics the behavior of the interpreter from the last section. We can change the right hand side of a single nonterminal and only affect the places where it is actually used. Thus, when combining grammars, we only work on the interaction points of the current and new grammar, not the entire existing grammar.

5 Generalized LL Parsing with Truffle

We saw how to systematically generate a recognizer for a grammar by hand. We wrote an interpreter that exhibits the same behavior but works for different grammars and even allows us to change the grammar. Using Truffle, we turned that interpreter into a just-in-time compiler. In this section, we discuss doing the same thing to a generalized LL parser. This is not a detailed explanation of generalized LL parsing itself. We will only highlight important differences to the grammar interpreters we saw before.

We started with an existing code base. Tillmann Rendel implemented a generalized LL parser based on the work of Scott and Johnstone [10]. Scott and Johnstone describe how to construct a parser from a grammar, similar to what we did in Section 2. Rendel’s implementation is a grammar interpreter, written in Java, similar to the interpreter in Section 3. Its source code is available at `https://github.com/Toxaris/gll`. For this thesis, we attempt to do what we did in Section 4, namely fork the interpreter and modify it to work with Truffle.

We chose to work with an existing code base, because GLL parsing is difficult. Writing a correct implementation is not a trivial amount of work. Rendel’s implementation is, unlike what Scott and Johnstone originally describe, an actual parser, not just a recognizer. Instead of a parse tree, it produces a Tomita-style shared packed parse forest (SPPF) [13]. The SPPF is a data structure that can be used to represent the kinds of parse trees that occur when parsing with ambiguous grammars. It uses maximal sharing to avoid exponential blow up. Our Truffle-enabled GLL parser uses the same data structures unchanged.

Rendel’s GLL parser is a grammar interpreter already, which appeared to work in our favor because Truffle works with AST interpreters. Unfortunately, the grammar representation was not actually tree shaped but a directed graph, including cycles and nodes with multiple parents. We avoided multiple parents and broke cycles by duplicating terminal symbols and wrapping nonterminal symbols in Truffle's call targets, as before. We made the AST nodes extend the `Node` class and annotated children as we saw in Section 4. According to the Truffle documentation, one should never hold a reference to an AST node directly, except to a node's own children, which are marked by annotations. In GLL we do that all the time. AST nodes are referenced, for identification purposes, in the SPPF, GSS (see below), and other AST nodes that are not children. This might be a problem when rewriting the AST, which we, for the most part, do not do. Everything seems to work fine for now, but this is something that should be fixed. However, this is a very unforeseen restriction, from the point of view of the original implementation. In retrospect, we might have been better off starting from scratch, ignoring the SPPF at first, but focusing on getting the Truffle-related aspects right.

Control flow in the recursive descent recognizer was simple. Start with the start symbol. When we encounter a nonterminal symbol, we call it, which amounts to a push to the call stack. Parsing a terminal symbol either succeeds or fails, either way we pop the stack and continue. The same applies when we reach the end of a sequence or run out of alternatives, with the appropriate return values.

Control flow in GLL parsing is more difficult. Where we have a simple call stack in our recognizer, the GLL parser uses a graph structured stack (GSS). In contrast to a simple stack, a stack frame in the GSS can have multiple children *and* multiple parents. When we parse a nonterminal in GLL, we parse, conceptually, all alternatives at once. Since a nonterminal can occur more than once on a right hand side, we deliver results to more than one parent stack frame at the same time.

There is only one Java call stack and it is limited. To simulate the above behavior and infinite stack space there is a trampoline in the GLL parser. Instead of calling into functions, the GLL parser registers a process object with the trampoline, to be executed later. If you are so inclined, you can think of the process objects as continuations. There are two nested main loops in the GLL parser, as seen in Figure 13. The outer is over every character of the input stream. The inner is over the set of active processes that need to still run with the current input character as their input. There is a set of future processes, that become the set of active processes when we parse the next input character.

Processes are call targets in our Truffle-enabled GLL parser. Unfortunately, we create processes at runtime, depending on the grammar and the input string. This is quite terrible for Truffle, because Truffle expects the AST to be stable, because it does partial evaluation on the tree. If the tree is not actually static data (after some rounds of rewriting, optionally), there is no point to partial evaluation. The grammar AST itself is of course

```
public void parse(final Reader reader) throws IOException {
  state = new ParsingState();

  state.start = start;
  state.active.add(Truffle.getRuntime().createCallTarget(
        new StartProcessRootNode(new Initial(), start)));

  int codepoint;
  do {
    codepoint = nextToken(reader);
    state.nextToken(codepoint);

    while (!state.active.isEmpty()) {
      final CallTarget current = state.active.poll();
      current.call(state, codepoint);
    }
  } while (codepoint >= 0);
```

Figure 13: Main loops of the Truffle-enabled GLL parser.

reasonably static, unless we change it, for which we have guards, like we saw in Section 4. We tried to eliminate some of the processes by doing work immediately, instead of registering it with the trampoline. Unfortunately, we were not able to eliminate all of them, especially not the future processes, which are supposed to be run when parsing the next token. With some clever grammar analysis, it might be possible to fix the set of processes. The GSS should also depend on the grammar, at least partially. For now, we have to treat it as entirely dynamic, meaning Truffle will not do anything to speed it up.

Truffle should be able to eliminate virtual calls that determine whether we are dealing with a terminal symbol or nonterminal symbol, for example. However, the normal JVM just-in-time compiler probably does the same thing. We can not expect impressive performance from our Truffle-enabled GLL parser. Not without some more research into analyzing grammars to predict control flow.

6 Study 1: Recursive Descent Recognizers

Implementing the Truffle-based grammar interpreter from Section 4 was a proof of concept experiment to gather experience with using Truffle for implementing parsers. In this section we describe the accompanying exploratory performance study. We will see the handwritten recognizers from Section 2 and the Truffle-enabled interpreters from Section 4. This study

$$
\begin{aligned}
\langle S \rangle \quad & ::= \ \texttt{EOF} \\
& \ \mid \ \langle C_0 \rangle \\
\langle C_i \rangle \quad & ::= \ \langle C_{i+1} \rangle \qquad \forall\, 0 \le i < n \\
\langle C_n \rangle \quad & ::= \ \langle E \rangle \\
\langle E \rangle \quad & ::= \ \text{'a'} \ \langle S \rangle
\end{aligned}
$$

(a) Loop on the inside.

$$
\begin{aligned}
\langle S \rangle \quad & ::= \ \texttt{EOF} \\
& \ \mid \ \langle C_0 \rangle \ \langle S \rangle \\
\langle C_i \rangle \quad & ::= \ \langle C_{i+1} \rangle \qquad \forall\, 0 \le i < n \\
\langle C_n \rangle \quad & ::= \ \langle E \rangle \\
\langle E \rangle \quad & ::= \ \text{'a'}
\end{aligned}
$$

(b) Loop on the outside.

Figure 14: Chained nonterminals grammars for the language a^*.

gives a rough idea of what kind of performance gains to expect. The results are surprisingly positive. The Truffle-based interpreters with the inline cache optimization appear to outperform the handwritten recognizers.

6.1 Goals

With this first study we establish an upper bound for expectations regarding the application of Truffle towards parsing. We perform multiple experiments designed to show off Truffle at its best. If Truffle does not do well in this, there would be no reason to expect it to perform well in the more involved case of GLL parsing.

We use a simple (regular) language, namely a^*, for comparing multiple recognizer algorithms and grammars against each other. Both grammar variants in Figure 14 contain a chain of nonterminals. At the end of the chain we parse a single 'a' and start again from the beginning. The grammars differ on where the recursion happens. In the grammar from Figure 14a the recursion is at the end of the chain. In the grammar from Figure 14b the recursion happens after we return from the chain.

We compare the Truffle-enabled grammar interpreter from Section 4 against handwritten parsers in the style of Section 2. Some of the run time will fall upon extracting a character from a string and comparing it to the reference terminal symbol. We use these convoluted grammars to inflate the effect of nonterminals have on parse time. The parsers interesting differences relate to nonterminals, as does the inline caching for lookups in the optimizing Truffle variant.

The chained nonterminals should also give us an idea about the cost of abstraction. Grammar composition happens on nonterminals, so additional nonterminals are extension points. We aim for unused extension points to be very cheap, performance-wise.

```
boolean s() {
  return eof() || c0(); }
boolean c0() { return c1(); }
boolean c1() { return c2(); }
[...]
boolean c200() { return e(); }
boolean e() {
  return character('a') && s(); }
```

(a) Loop on the inside.

```
boolean s() {
  return eof() || (c0() && s()); }
boolean c0() { return c1(); }
boolean c1() { return c2(); }
[...]
boolean c200() { return e(); }
boolean e() {
  return character('a'); }
```

(b) Loop on the outside.

Figure 15: Handwritten recognizers for the chained nonterminals grammars.

```
public static boolean optimal(String s) {
    final int length = s.length();
    for (int i = 0; i < length; i++)
        if (s.charAt(i) != 'a')
            return false;
    return true;
}
```

Figure 16: Perfect recognizer for a^*.

6.2 Setup

In this study, we use two variants of the grammar from Figure 14 for the language a^*. One with the loop at the inside (Figure 14a) and one with the loop at the outside (Figure 14b). We compare four implementations:

- First, handwritten recognizers in the style of Section 2. There is one for each variant of the grammar with chain length 200. See their implementation in Figure 15.

- Second, the Truffle-enabled grammar interpreter from Section 4 using indirect calls.

- Third, the same Truffle-enabled grammar interpreter, this time using the inline cache with direct calls.

- Fourth, an "optimal" recognizer for a^*, which is a loop over the input string that checks that every character is an 'a', as seen in Figure 16.

We run three distinct benchmarks:

Hardware	Specs
CPU	Intel(R) Core(TM) i5-4670 CPU @ 3.40GHz
RAM	16GB DDR3
Software	**Version**
Operating system	Linux 3.16.0-2-ARCH #1 SMP PREEMPT Mon Aug 4 19:04:45 CEST 2014 x86_64
Java Virtual Machine	OpenJDK 64-Bit Server VM (build 25.0-b63-internal-graal-0.3, mixed mode)
Truffle	0.3 (binary release bundled with JVM)
Scalameter	0.5-M2 (for Scala 2.11.x)
Scala	2.11.0

Figure 17: Hardware and software used for benchmarking.

- First, we compare the two variants of the Truffle-enabled grammar interpreter, with direct and indirect calls on the loop on the inside variant of the grammar, for chain lengths 1, 50, 100, 150, and 200.
- Second, we compare the same parsers on the loop on the outside variant of the grammar for the same chain lengths.
- Third, we compare the handwritten recognizers, which use the chain length 200, and the "optimal" recognizer, which is the same for all chain lengths.

We use Scalameter for benchmarking. Each benchmark instantiates a new Graal/Truffle-enabled JVM using the following command line options: `-server -Xss64m -G:TruffleCompilationThreshold=1`. We rely on Scalameter to warm up the just-in-time compiler for the normal Java code and increased the minimum number of warmup runs to 1000. For the Truffle-based interpreters, we run our own warmup loop that parses the test string 10000 times to give Truffle time to make sure the AST has been rewritten, Graal has been invoked, and we actually use the optimized code. We make sure to keep the ASTs stable between warmup runs and use the same ASTs for the actual measurements. After warmup, we parse the input string a^{150} 10000 times with every variant in an AABB scheme and measure runtime.

Benchmarks were run on an otherwise idle desktop computer. See software versions and hardware specs in Figure 17. Find the benchmark code together with the benchmarked code at `https://github.com/fehrenbach/parsers`.

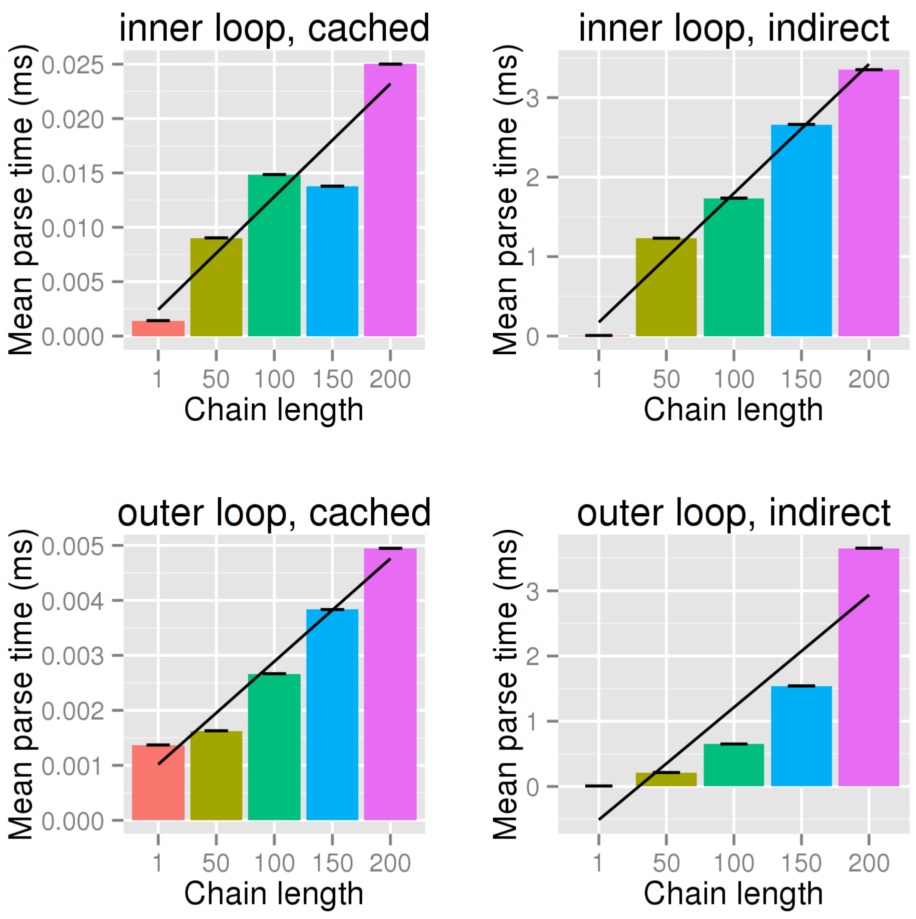
6.3 Data

Figure 18 shows the results of the first two sets of benchmarks. There are four distinct plots, for the four combinations of grammar variant and whether direct or indirect calls were used by the interpreter. The headline above each plot lists the particular combination where *inner loop* means we use a grammar with the loop on the inside as in Figure 14a; *outer loop* means we use a grammar with the loop on the outside as in Figure 14b; *cached* means the interpreter caches the lookup of nonterminals and uses guarded direct calls; and *indirect* means the interpreter performs a lookup and indirect call for every nonterminal. On the x axis we see the different chain lengths. Each bar's height represents the mean parse time in milliseconds for the string a^{150}. Note that each plot has its own y axis. The small horizontal black line at the top of each bar is the 95% confidence interval of the mean. The confidence intervals are so tight that they appear as single black lines only. The slanting black line in each plot is a linear regression line. Their equations can be found in the table below the plots in Figure 18. We use all of the data points for the linear regression, not only the means.

Figure 19 shows descriptives, quantil-quantil plots, and box plots for the handwritten and optimal parsers from our third experiment, as well as for the direct call interpreter versions with chain length 200 of the previous two experiments. The text above each table of descriptives indicates which parser's data the respective row shows. For example, the first row shows the descriptives, Q-Q plot, and box plot for the parse time in milliseconds for the handwritten recognizer from Figure 15a. The descriptives list the minimum, first quartile, median, arithmetic mean, third quartile, and maximum time, in milliseconds, to parse the string a^{150}.

The box plots on the very right of each row are hardly recognizable as such. They show that we have many measurements very close to each other with a couple of heavy outliers. Since the box plots do not give a good indication of the distribution of measurements, we created Q-Q plots against a normal distribution, found in the middle of each row. If the measurements were a "perfect theoretical" normal distribution, all black points would lie on the blue line. Again, we see the outliers that represent a much higher parse time than what we would expect if the data followed a normal distribution. Except for the outliers, most of the data follows a normal distribution rather well. However, the first plot shows many outliers and seems to be a bit skewed to the right. The second plot indicates that the measurements are skewed to the left compared to a normal distribution, that is, there are a couple more very short parse times than what we would expect. The third, fourth, and fifth plots are almost perfect. Although, in the fifth plot, the steps might indicate that we actually deal with three distinct populations.

Figure 20 shows the mean parse time of the five fast parsers as bars next to each other. This does not show anything we did not see in Figure 19

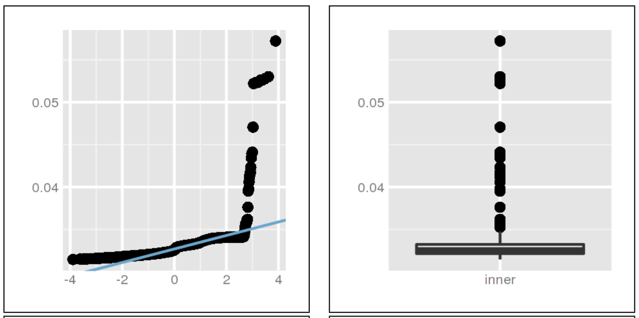


Variant	Linear regression equation (n = chain length)
inner loop, cached	$0.002369 + 0.0001043n$
inner loop, indirect	$0.1643 + 0.01630n$
outer loop, cached	$0.001004 + 0.00001881n$
outer loop, indirect	$-0.5234 + 0.01733n$

Figure 18: Mean parse time against chain length for different parsers and grammar variants.

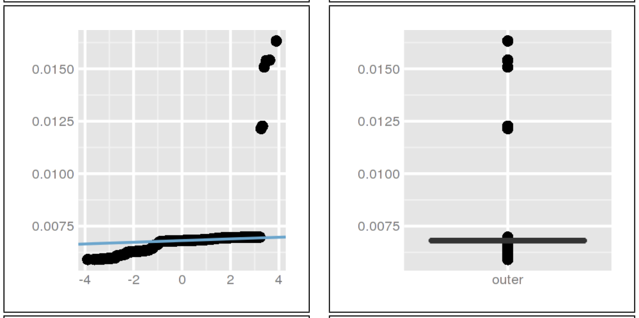
Handwritten,
loop on the inside

Min.	0.03146
1st Qu.	0.03216
Median	0.03258
Mean	0.03279
3rd Qu.	0.03324
Max.	0.05722



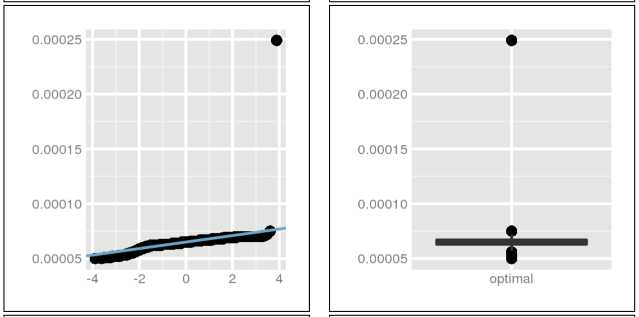
Handwritten,
loop on the outside

Min.	0.005912
1st Qu.	0.006784
Median	0.006810
Mean	0.006764
3rd Qu.	0.006838
Max.	0.016328



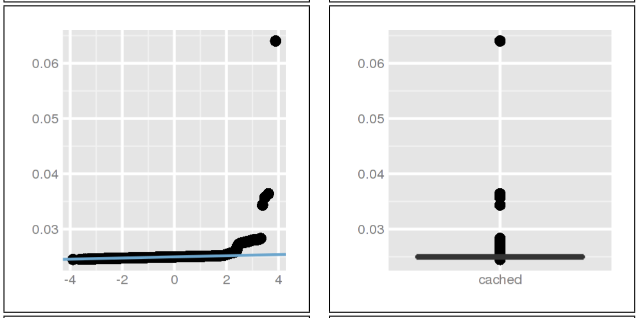
Optimal parser

Min.	6.300e-05
1st Qu.	6.500e-05
Median	6.482e-05
Mean	6.700e-05
3rd Qu.	2.490e-04
Max.	0.06403



Direct call Truffle,
loop on the inside

Min.	0.02451
1st Qu.	0.02491
Median	0.02498
Mean	0.02501
3rd Qu.	0.02505
Max.	0.06403



Direct call Truffle,
loop on the outside

Min.	0.004746
1st Qu.	0.004880
Median	0.004911
Mean	0.004946
3rd Qu.	0.005013
Max.	0.009586

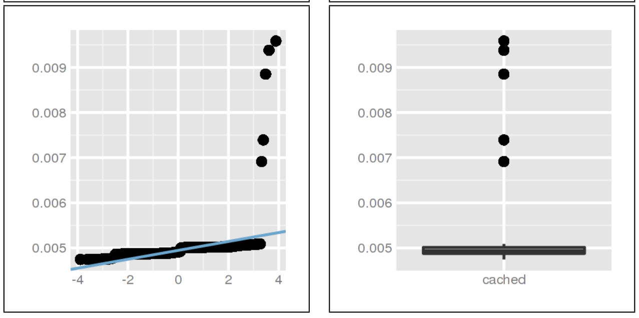


Figure 19: Q-Q plots, descriptives, and box plots for the fast parsers with chain length 200.

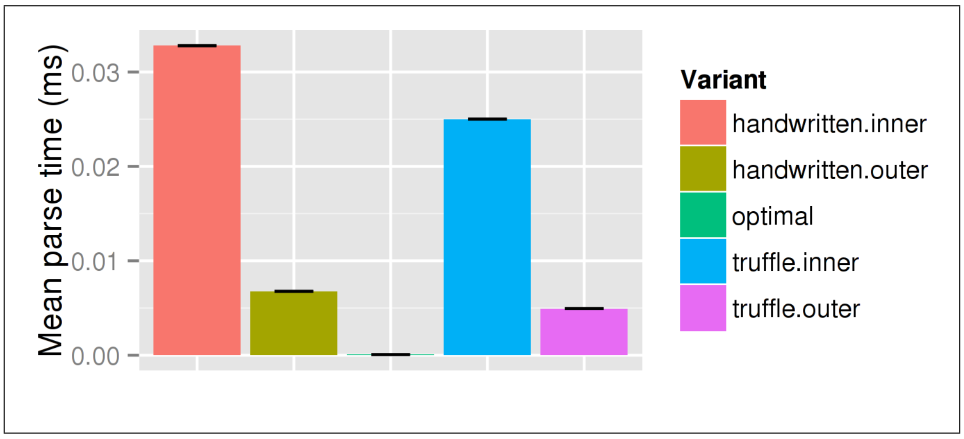


Figure 20: Mean parse times for the fast parsers with chain length 200.

already, but makes comparing the five fast parsers against each other easier. We see that the variants of the grammar with the loop on the outside are significantly (the confidence intervals clearly do not overlap) faster than the loop on the inside variants for both the handwritten parsers and the caching Truffle-enabled grammar interpreters. Comparing the handwritten recognizers against the Truffle-enabled grammar interpreters, we see that the interpreters are actually significantly faster for the respective grammar variants but not so fast as to be consistently faster irrespective of the grammar variant. The optimal recognizer from Figure 16 is clearly faster than even the fastest recursive descent recognizer.

6.4 Interpretation

Barring gross mistakes during benchmarking, for a discussion of which see the next section, the data seems to back up our hope that Truffle can be used on grammar interpreters with reasonable results. In fact, for this particular grammar, the Truffle-based interpreters actually outperform recognizers that people would conceivably write by hand. And this is with interpreters that do more work than the handwritten recognizers, because those do not support changing the grammar and therefore need not and do not check guards that the grammar is unchanged. This is quite surprising and far exceeds our hope that the just-in-time compiled interpreters might be within one order of magnitude slower than an equivalent handwritten recognizer.

We do not know exactly why the Truffle-based interpreter is this fast. Truffle is not forthcoming about what it does internally and reading the generated x86 assembly is beyond our ability. Our best guess is that the partial evaluator eliminates function calls, as it is meant to do. From the limited amount of debug information we get out of Truffle, it seems that it inlines 18 nodes at a time. For chain length 200, that would amount to only 12 actual function calls, compared to the 200 function calls that the handwritten recognizer performs. This could offset the 200 guards we need to check, especially since those are designed to be very fast if their assumption holds.

We can interpret the chain length experiment in various ways. On the one hand, we see that the cost of abstraction, as measured by additional steady-state runtime for each nonterminal, is negligibly small for the optimizing just-in-time compiling interpreters. If we take the linear model literately, it is somewhere around 0.0001043 or 0.00001881 milliseconds per nonterminal for a string of length 150. The grammars for some small languages, like SQL dialects, consist of about 200 nonterminals. This would mean that one extension point (one additional nonterminal that only does forwarding) would cost us a couple of picoseconds per character parsed.

On the other hand, parse time certainly depends on the size as well as the structure of the grammar. It takes less time to parse our test string with the loop on the outside, chain length 200 grammar, than it takes to parse it using the loop on the inside grammar variant with only a chain of length 50. Grammar optimization on the AST level might yield order of magnitude improvements. We can think of the optimal parser from Figure 16 as the end of that particular road.

6.5 Threats to Validity

We measure wall-clock times well below microseconds in some cases. Timing resolution should not be a problem but scheduling and migration of processes between CPU cores can be a problem. We compensate with a high amount of individual measurements. The bar plots and tight confidence intervals indicate that we are not heavily hit by scheduler issues. The Q-Q plots indicate that there is a large normally distributed portion of our data which usually means we collect enough measurements to have the law of large numbers apply.

There are plenty of environmental influences we do not control and that persist across one JVM invocation. They might explain the somewhat surprising top-right graph in Figure 18. There we see that the 150 nonterminals chain grammar is actually faster than the 100 nonterminals chain grammar. The tight confidence intervals indicate that this was actually the case for this JVM invocation, not some random scheduler interruption. We do not know the reason for this, it might be some alignment issue with some cache boundary or something else entirely. In a future study, we would like to run benchmarks in multiple JVM invocations, across more machines and maybe reduce the number of individual measurements to keep the overall experiment duration feasible.

We only measure steady-state performance, that is parse time after the just-in-time compiler did its work. In a way this is unreasonable, since our original target use case is parsing files whose language changes during parsing. However, compilation time almost entirely depends on Truffle internals since we do not yet do any elaborate analysis during AST rewriting. This means, if Truffle turns out to be slow to kick in, we can do little about that. It also means, that we benefit from all the work that is done on Truffle. If there should be issues with start up performance of Truffle-based interpreters, there are just-in-time compilation experts who will be addressing them in Truffle and we will benefit from their work immediately.

The plots that compare the handwritten recognizers against the optimizing Truffle-based interpreters do not compare equals. Truffle-generated code is compiled by Graal whereas the handwritten recognizers and the optimal parser are regular static Java code that is compiled by the default compiler back end in the JVM. Someday, Graal might become the default for any Java code. Until then, this comparison makes sense because most people will only ever use the current default compiler.

Almost every set of measurements has at least one outlier that is more than double the average, and even third quartile, parse time. This might be the first measurement in a series, where the actual machine code is not yet in the instruction cache. Scalameter does not make this information readily available. We hope that in the future, Scalameter will do the reasonable thing, which is to discard the first measurement [7].

Throughout this section, we dealt with recognizers, not parsers. We do not construct parse trees, but only solve the word problem, does this string match our grammar or not. For this exploratory study, this is fine. We intentionally constructed an interpreter where it should be easy for Truffle to shine. This also means that these results can not immediately be generalized to every kind of grammar interpreter. Still, we see that Truffle can be used successfully on something that is not obviously a programming language interpreter.

7 Study 2: Generalized LL Parsing

Truffle did well on the simple recursive descent recognizers we saw in the first study. With this second study we aim to discover how well Truffle does with the significantly more complicated GLL parser.

7.1 Goals

In this study, we compare two GLL implementations against each other. The first was written in Java and is a grammar interpreter already. The second shares almost all of its code, notably the parse forest data structures and the graph structured stack. It was modified to work with Truffle, as detailed in Section 5.

Even though most of most programming languages syntax is context-free, syntax definitions are mostly in grammar formalisms that have more features than basic context-free grammars. Both of our implementations "only" support context-free grammars. They do not have special support for disambiguation or scannerless parsing like for example the SDF parser SGLR does. As a consequence we did not attempt to parse any real language. Instead, we reuse several context-free grammars, that are already used for testing, for benchmarking. In addition, we use the chained nonterminal grammar with the loop on the inside from Figure 14a.

These grammars are not representative of the kind of real world programming language grammars we would like to parse. However, since our two parsers share most of their code, these tests should give an impression of the effect Truffle can have.

Again, we do not measure start up performance. This seems contrary to the SugarJ setting we described in Section 1. However, we are reasonably certain that a grammar interpreter does much better than the original parser with the ahead-of-time generated parse table for parsing a single file. It is more interesting to us whether a Truffle-based interpreter is faster, slower, or as fast as a regular grammar interpreter.

7.2 Setup

Unless otherwise noted, we use the same setup as in our first study. See Section 6.2 for details.

We use two GLL implementations:

- The first is a plain Java implementation that can be found at `https://github.com/Toxaris/gll`. We only changed this implementation to also support parsing multiple streams with one instantiation of the grammar, as the Truffle-enabled parser does. Find the changes and the benchmark code at `https://github.com/fehrenbach/gll` on the branch `for-upstream`.
- The second parser is a fork of the first parser that was modified to work with Truffle. The two implementations share all of the data structure code and most of everything else. Find the fork at `https://github.com/fehrenbach/gll`.

We performed four different benchmarks with each parser. Every benchmark starts a new JVM and does warmup as described in Section 6.2.

The first benchmark uses the chained nonterminals grammar with the loop on the inside that we already saw in the first study. As before, we use the chain lengths 1, 50, 100, 150, and 200 and string to be parsed is a^{150}. We

```
⟨S⟩  ::= ϵ | [a-Z] | ' '                ⟨P⟩  ::= '(' | ')'
       |  ':' | ':' ⟨P⟩
       |  '(' ⟨S⟩ ')'
       |  ⟨S⟩ ⟨S⟩
```

(a) Balanced smileys grammar.

#	String
1	((:))
2	(((((((((((((((((((())))))))))))))))))))
3	((((((((((((:))))))))))((((((((((:())))))))))))
4	((((((((((((:))))))))))((((((((((:)))))))))))))
5	(((((((((())))))))))
6	((((((((((:)))))))))) ((((((((((:)))))))))) ((((((((((:)))))))))) ((((((((((:)))))))))) ((((((((((:)))))))))) ((((((((((:))))))))))
7	((((((((((:)))))))))) ((((((((((:)))))))))) ((((((((((:)))))))))) ((((((((((:)))))))))) ((((((((((:)))))))))) ((((((((((:)))))))))))
8	((((((((((:))))))))))
9	((((((((((:)))))))))))
10	((:):::(()()):)(()()():())aaa)(:(a:)a:((())a(((a(:())aa():a:)((()):)(()(:)(a())a:()a)a():(
11	()(((a)((aa)))a)a()(a)(aa:a)()(((:())aa)):()():():a:(a)(a())a:)::a:(aa:):()((a:)())aa)a(a:)
12	()((:a(a()()a))())((:a(:a)(()a((((a((a(()(:aa()()())))):)(():):)(:(a))():(())(():()):):(()a))
13	():)((()():(:()))))::aa((((:(((:)))::a:(:))()a)):(a):::((()a((a(aa(():))(():())((::a)a)):)()
14	():)a((a:((aaa(()))((((()a))()))a(:)):)a((:())(a:(:):((a(:(::())a()())::()a)(a)):((aa)a(:(())
15	()a(:)(a:a):(())):a()():((a(:):a()()::)(a:)(()a((a:)(a)a(a:a:)(a)a(a:(()()()::a()a()(()a:())))
16	(:)
17	(::a((a)a:()):):a)aa:)a(:::))(a())aa(a():))(:)a)((():)(:a:)a))):a(a)((:()(()())a))()a((()a))
18	(a((f((g(((g((:))))g))))))::((((((((((((((((((((:))))))))))))))))))))((:))
19	(a((f((g(((g((:))))g))))))::((((((((((((((((((((:))))))))))))))))))))((:)))
20	:(a):(:)aa)a(:()::():))a:aaa:)(:)((()()))a()((((()(:)))(:(aa:()())())a((a)a:(:()))(a((():)))
21	:)()((a)):(():a:a:)(:a)):)(()(:)::::(a(::a())(a):(:((((:(aa(()))a)((((((((()a()a):)))((:))))))))))
22	::((:))(((:)(aaa)(a())()(a:)(:)(:)()):)a())aa)())(():a):()::):)a()())a()):):(:a)a):()(a)(a)
23	:a:)(:))()(()()a)aaa::a()()a:()()a::)((()(a(a))))try implementing sleep sort if you are stuck:(:)a)
24	a(a)::(((::)))())((a)(:((:a())):((::(:()(a)))i am trapped in a test case generator :(:(a(:::))
25	hacker cup: started :):)
26	hello world
27	i am sick today (:()

(b) Test strings.

Figure 21: Balanced smileys benchmark data.

perform 1000 measurements with each chain length and implementation. Each parser call also reinitializes the internal parser state so we can use the same grammar and do not have to warm up the Truffle AST.

The second set of benchmarks is about balanced parentheses and smileys. The language described by the grammar in Figure 21a contains only strings where opening and closed parentheses match, except that it does not consider the parentheses in the smileys `:)` and `:(`. We used the strings in table 21b which should all be elements of the language. We configured Scalameter to perform 100 measurements for each test string.

The third set of benchmarks uses the grammar Γ_2 from the original paper on GLL [10]. This grammar is a very ambiguous one for the language b^+ as seen in Figure 22a. We parse the strings $b, bb, bbbb, \dots, b^{64}$ 100 times each.

The fourth set of benchmarks uses a variant of the grammar from before.

$$\begin{aligned}\langle S\rangle \quad &::= \text{'b'}\\ &\mid \quad \langle S\rangle\ \langle S\rangle \mid \langle S\rangle\ \langle S\rangle\ \langle S\rangle\end{aligned}$$

(a) Very ambiguous grammar (Γ_2 [10]).

$$\begin{aligned}\langle S\rangle \quad &::= \text{'b'} \mid \langle S\rangle\ \langle S\rangle\ \langle A\rangle\\ \langle A\rangle \quad &::= \langle S\rangle \mid \epsilon\end{aligned}$$

(b) Factored grammar (Γ_2^* [10]).

Figure 22: Two variants of a grammar for the language b^+.

In the variant in Figure 22b the nonterminal $\langle A\rangle$ has been factored out. As a result, parsing should be faster. We parse the strings $b, bb, bbbb, \ldots, b^{64}, b^{128}$ 100 times each.

7.3 Data

Figure 23a compares the Truffle-enabled parser to the existing plain Java GLL parser. The Truffle-enabled parser is always on the left and in a red color. The original GLL parser is on the right and colored turquoise. We omitted confidence intervals because they are as tight as we saw in Study 1.

The first two plots report mean parse time for the chained nonterminals grammar and the balanced parenthesis with smileys grammar. Lower bars are better, and we see that the original plain Java GLL parser is faster for every test string.

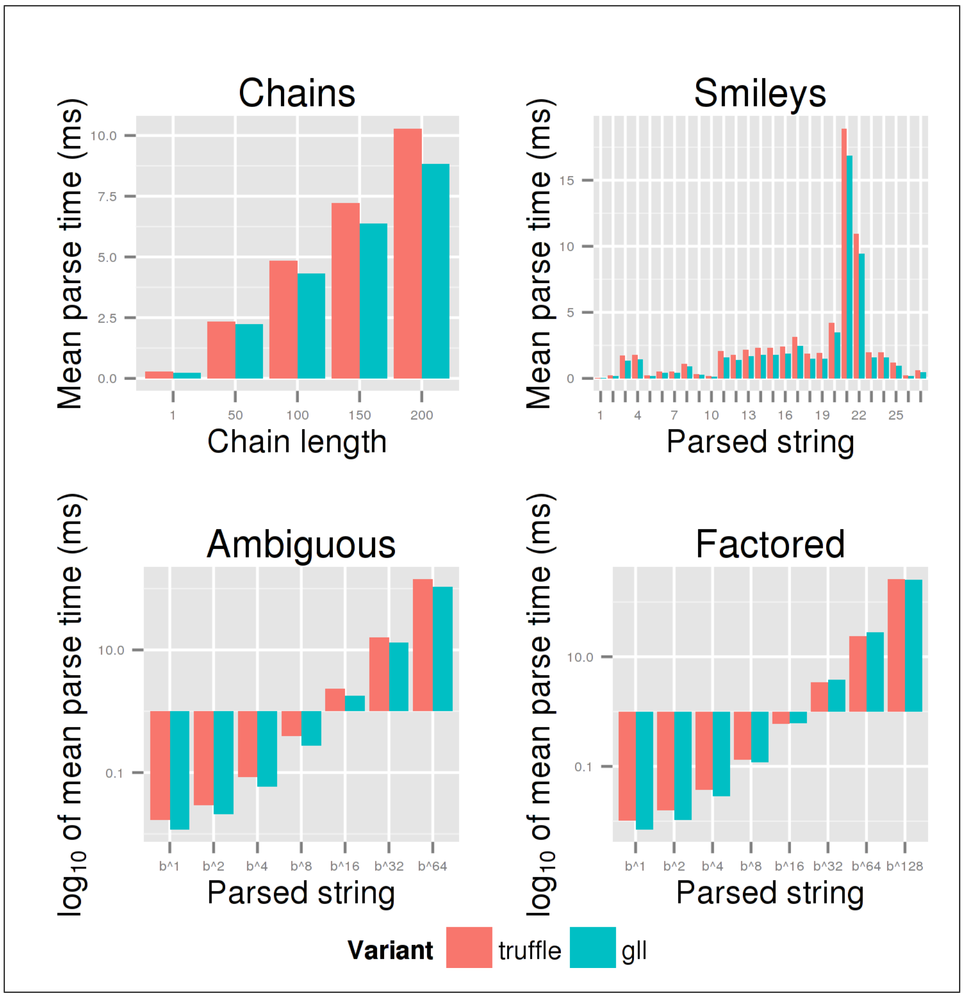
The second two plots are log scale. Again, lower bar is better. For the downwards-pointing bars on the left side of each plot, the taller bars are actually better.

The Truffle-enabled parser does better than the original plain Java GLL parser in only three benchmarks, namely the test strings b^{16}, b^{32}, and b^{64} when parsed with the factored grammar from Figure 22b.

Comparing relative performance is difficult for the log scale plots and the many test strings in the smileys benchmark. We calculated the speedup the Truffle-based parser gives us compared to the plain Java GLL parser by dividing the means for every test string in every set of benchmarks for the two parser variants. Find the descriptives in the table in Figure 23b. Below 1.0 means the original plain Java GLL parser is faster. Above 1.0 means the Truffle-enabled parser is faster.

As expected, only the factored grammar benchmark has any speedups above 1.0. The highest is 1.1550, that means the Truffle-based parser was on average 15% faster than the original plain Java GLL parser for one set of test strings, namely b^{64}.

Across all test strings with all four grammars, the Truffle-based parser is on average about 20% slower. At the worst, it is a good 30% slower and it is up to 15% faster for some select test strings with the factored grammar.



(a) Benchmark results for the existing plain Java GLL parser and modified Truffle-enabled parser.

Benchmark	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Chains	0.8370	0.8576	0.8834	0.8838	0.8902	0.9509
Smileys	0.7676	0.7832	0.7929	0.7996	0.8073	0.8918
Ambiguous	0.6948	0.7022	0.7040	0.7372	0.7596	0.8382
Factored	0.6767	0.7494	0.9403	0.9144	1.0470	1.1550
Overall	0.6767	0.7697	0.7988	0.8188	0.8376	1.1550

(b) Speedups (below 1.0 means original plain Java GLL parser is faster).

Figure 23: Second study results.

7.4 Interpretation

The Truffle-enabled GLL parser is slower than the original plain Java implementation. This does not come as a surprise, exactly. Truffle seems to do best when there is a lot of computation and control flow that can be eliminated by partial evaluation. Our GLL implementation does little actual computational work for every symbol in the grammar. We removed some of the dynamism by doing work immediately, instead of scheduling it to be done later. It does not seem to be enough.

A 20% slow-down seems pretty bad, but fortunately we have barely scratched the surface on what could be done in a GLL parser. The only AST rewriting we do, is caching nonterminal lookups, which are only one virtual call in the original GLL parser. We trade that for repeated indirect calls into Truffle code, and we saw in the first study that this might not be a good idea. There is much work to be done in the area of (just-in-time) analysis and specialization of grammars. For one example, compare the ambiguous grammar with its factored version. Of course it is not as easy as just factoring the grammar. We deal with a real parser here, not just a recognizer, thus we would need to produce a parse tree (or forest) that is consistent with the unfactored grammar, even if we perform factorization behind the scenes.

We can also aim to help Truffle by making more of the control flow visible to the partial evaluator. The graph-structured stack (GSS) is partially dynamic and partially static data, which is notoriously difficult to deal with using offline specialization [9]. While Truffle claims to use online partial evaluation [14], it is still necessary for the language implementer to explicitly annotate static parts of the input, namely the AST. This has a distinct feeling of offline partial evaluation to it. However, it is possible to additionally perform AST rewriting to tell Truffle about dynamic parts of the input that should be considered static, at least until it changes, which must be guarded against to allow ford deoptimization. In the future, we look towards using this mechanism to discover the static part of the GSS by analyzing the grammar and rewriting to specialized nodes that convey this information to Truffle.

In summmary, there is a slow-down, but it is not terrible. If we are so inclined, we can choose to interpret it as warranting further research and engineering into grammar analysis before writing off the idea of just-in-time compiled parsers.

7.5 Threats to Validity

We do not use grammars of practical (programming) languages for benchmarking. This is because there is no scanner for our parser and it does not support scannerless parsing. However, our test grammars do exercise different parts of the parser. The chained nonterminal grammar does have as many productions as a small programming language. The ambiguous grammar is ambiguous. It shows that we at least did not break the part of the GLL algorithm that deals with ambiguity. Thus, we claim at least some external validity.

The two parsers share most of their code. We can be confident that we actually measure the difference Truffle makes. Keep in mind however, that part of the changes were about doing more work immediately instead of scheduling it to be done later, in the Truffle-enabled parser. In theory, this should make a big difference for Truffle, because it allows partial evaluation to actually span more than one AST node, and a small difference for the original plain Java parser.

We share some of the experimental setup with the first study. We discussed threats to validity for that as well, in Section 6.5. What was said there about measurement accuracy applies here as well, though to a lesser extent because parse times are in general higher. Everything else, about JVM invocations, compilers, and outliers applies to this second study also.

For the very close result in the factored grammar, b^{128} benchmark, we could have performed a t-test. However, since this is an exploratory study without a formulated hypothesis we chose not to. 95% confidence intervals of the means are too tight to be seen in the plots, so, by rule of thumb, the difference of means is statistically significant.

8 Discussion

We were able to answer some, but not all, of the questions we asked in the introduction. Our implementation work demonstrates that it is indeed possible to use existing just-in-time compilation technology, specifically Truffle, to implement parsers. This immediately answers another question: yes, we can start parsing with the grammar as it is, without first generating a parse table. We still deal with interpreters that can be run immediately. Machine code generation happens in the background but only after a number of executions anyway. Nonterminal resolution is dynamic so we can indeed incrementally modify the parser, keeping unchanged parts. With guarded speculative inline caching and deoptimization we even seem to be able to avoid some of the runtime overhead that one would suspect from the additional flexibility.

We did not measure startup performance in any of the performance studies. This may seem odd considering the setting that motivates this work. However, we already know that a grammar interpreter is much faster than an ahead-of-time generated parser for a single file. The obvious practical implementation would be to start parsing with the interpreter while ahead-of-time compiling a parser in the background and switching once that is done. There are much more interesting research questions in just-in-time compiled parsers.

We did not implement optimizations on the grammar level. In the second performance study (Section 7), we compared two variants of the same grammar against each other; the very ambiguous grammar Γ_2 and its factored variant Γ_2^*. This is a well-known optimization [10] and the factored grammar indeed performs much better. It parses b^{128} in the same time it takes to parse b^{64} with the original grammar. This kind of grammar rewriting seems to fit very well with the self-optimizing AST interpreter approach of Truffle [15]. Our performance study suggests, that even implementing only this one optimization sets off the performance overhead that Truffle incurs for some grammars. In the future, we would like to explore this kind of grammar optimization. There are some interesting questions in particular with regard to performing them just-in-time and on a possibly changing grammar.

If an optimization requires extensive static analysis, is it worth it? We already decided to not do parse table generation because the ahead-of-time costs are too great. Maybe it is possible to do the analysis for only a part of the grammar, the part that is heavily exercised by the input file. This is the kind of setting in which a just-in-time compiler can outshine an ahead-of-time compiler.

The simple inline cache optimization is guarded against changes to single nonterminals. How do grammar optimizations react to changes of the grammar? We still aim to keep as much of the parser as possible and only change the parts that depend on the changed parts of the grammar. However, changes to a single nonterminal may affect static analysis results for the whole grammar, FIRST sets are an example.

Our Truffle-based GLL implementation does not perform nearly as well as our Truffle-based simple recursive descent recognizer, even compared to their respective peers. We suspect this is because the control flow in GLL is largely opaque for Truffle. The problem seems to be the trampoline which acts as a scheduler for parsing processes. In Section 5 we mention possibly analyzing the grammar to recover static information about the control flow that we could then encode in the AST and thereby make available to Truffle. Perhaps Truffle could be extended to better handle this kind of control flow instead. As we see it, the situation in GLL is very similar to what a programming language with cooperatively scheduled green threads or coroutines would need. Many programming languages, Erlang, Go, and Concurrent Haskell come to mind immediately, offer a more lightweight concurrent programming facility than operating system-level threads. They include their own scheduler that manages threads of execution, like the trampoline in GLL. It seems that implementing such a language using Truffle would require some additional support from Truffle. If this should ever happen, we hope we could adapt the Truffle-based GLL accordingly.

Our second study suggests that the Truffle-based GLL implementation is barely competitive with another GLL grammar interpreter. The first study however suggests that a Truffle-based grammar interpreter can outperform a handwritten recognizer. Granted, the recognized class of languages (LL(0)) is limited. We would like to explore whether just-in-time compiled parsers for other language classes do equally well. Control flow seems to be an issue, but GLL and LL(0) are close to the extremes in either direction. There may be languages in between that still benefit from a Truffle-based interpreter. Perhaps a regular expression engine could do well when written with Truffle. The Linux kernel includes a just-in-time compiler for firewall rules that compiles to x86 machine code. We do not know what kind of language class those correspond to but Truffle might be suited for similar applications. Probably in user space code, not kernel code, though.

We use a very simple model of changing grammars. In our setting, a grammar only changes between parsing entire sentences. There is work on adaptive grammars that change during parsing of a single sentence (see also below). If we manage to address the problems in the GLL implementation, it would be interesting to explore whether it is possible to adapt it to other models of grammar change.

9 Related Work

We are certainly not the first to consider parsing as grammar interpretation. Parser generation is commonly used as an example in work on partial evaluation [9]. The use of Truffle for compiling parsers, however, is largely unexplored territory. Chris Seaton, who now works on Truffle, used to work on an extensible programming language called *Katahdin* [11]. It uses a extended parsing expression grammar algorithm for parsing. In the original implementation, the parser is just-in-time compiled by generating .NET bytecode which is then compiled to machine code by the .NET runtime. Seaton later attempted to reimplement the parser using Truffle, but he "didn't get far enough to measure any performance" [12].

Nowadays, people say a programming language has a context-free grammar, but in fact, people impose additional restrictions. For example, we parse any identifier, not only those that are in scope. The compiler or interpreter only checks that variables are bound later, after parsing is already done. The *ApeseLanguage* uses a dynamic parser called *ZZParser* [1]. It allows grammar rules to change the grammar when being parsed. Cabasino et. al. describe how they make the variable declaration rule extend the grammar to include a rule that allows to parse the variable that was just declared. Because there are different syntactic sorts for different types, the parser even does the type checking for them. Most of this kind of work seems to have been done in the 80s and 90s and research on formal languages has been rather dormant since then, unfortunately. Christiansen uses the term *adaptive grammars* in a survey [2] that discusses the goals and limitations. The underlying motivation is to describe more of the syntax of a language in its grammar. Christiansen further shows, that the same mechanism can be used to syntactically extend the language. The example he uses is a new data type for complex numbers. It includes syntax to access the imaginary and real parts of a complex number. Interestingly, this new syntax interacts with the existing syntax. The programmer can use the real or imaginary part of a complex number wherever a real number is expected. This is remarkably similar to some of the simpler language extensions in SugarJ [4]. The ApeseLanguage allows programmers to define their own operators using this mechanism. It is not clear, however, whether the same technique can be applied to cover the full extent of a sugar library's capabilities.

Heering et. al. [8] start within roughly the same setting. They attempt to reduce the overhead of parser generation for frequently changing grammars in an ASF/SDF integrated development environment. In some ways, ASF/SDF is a predecessor to SugarJ, and the challenges for a parser are similar. They design a lazy, incremental parser generator. Lazy means that the parser is only constructed when needed and only as far as needed. No code is generated for unused nonterminals, for example. Incremental means that changing the grammar only changes those parts of the parser that correspond to the changed parts of the grammar. Our decision to guard against grammar changes on the level of individual nonterminal definitions and our cache invalidation strategy mirrors Heering's model of incremental change. This may be inherent in our similar model of grammar change, namely one top-level declaration at a time, unlike the work on adaptive grammars, where the scope of grammar changes has to reflect the scope of variable bindings. While they solve a very similar problem, the technical realization that Heering et.al. offer is quite different from ours. They use a table driven LR parser and lazily generate the parse table. We use Truffle to just-in-time partially evaluate a grammar interpreter. Their solution seems to be good and we wonder why this has not been adopted for SGLR, the Scannerless GLR parser that is used by SugarJ, Stratego/XT, the ASF+SDF meta-environment, and others, to parse according to SDF grammars.

10 Conclusions

We show that Truffle can be used to implement parsers. We implemented Truffle-based, just-in-time compiling grammar interpreters for simple recursive descent recognizers of LL(0) languages and context-free language parsers using the GLL algorithm. Exploratory performance studies suggest that the simple recognizers could actually outperform handwritten recognizers. The Truffle-based GLL implementation is slower than the original grammar interpreter. However, the slowdown is well below an order of magnitude.

The GLL parser is not yet ready for use in SugarJ. We present some evidence that with some effort on grammar optimization it might be in the future. Truffle-based grammar interpreters for other language classes might already be competitive with handwritten solutions. If you have a performance sensitive problem that might conceivably formulated as an interpreter, we suggest you look at Truffle first.

Acknowledgments

I would like to thank the helpful people on the Truffle mailing list, especially Christian Humer and Chris Seaton. Prof. Ostermann made it possible for me to attend SPLASH'13 where Sebastian Erdweg made me aware of Truffle. I owe much to my supervisors Tillmann Rendel and Paolo Giarrusso for their help and helpful discussions, with and about Scalameter, GLL, partial evaluation, and personal motivation crises. Last but not least, thanks to Martin Salzer for counseling on statistical methods.

References

[1] S. Cabasino, Pier S. Paolucci, and G. M. Todesco. Dynamic parsers and evolving grammars. *SIGPLAN Not.*, 27(11):39–48, November 1992.

[2] Henning Christiansen. A survey of adaptable grammars. *ACM SIGPLAN Notices*, 25:35–44, 1990.

[3] Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2011.

[4] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406. ACM, 2011.

[5] Stefan Fehrenbach, Sebastian Erdweg, and Klaus Ostermann. Software evolution to domain-specific languages. In *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 96–116. Springer International Publishing, 2013.

[6] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45–50, 1999.

[7] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[8] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 179–191, New York, NY, USA, 1989. ACM.

[9] Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, September 1996.

[10] Elizabeth Scott and Adrian Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 253(7):177–189, September 2010.

[11] Chris Seaton. A programming language where the syntax and semantics are mutable at runtime. Master's thesis, University of Bristol, 2007.

[12] Chris Seaton. Message to the Truffle mailing list. `http://mail.openjdk.java.net/pipermail/graal-dev/2014-July/002393.html`, July 2014.

[13] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[14] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 187–204, New York, NY, USA, 2013. ACM.

[15] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.