ABSTRACT
Grammars for programming languages are traditionally specified statically. They are hard to compose and reuse due to ambiguities that inevitably arise. PetitParser combines ideas from scannerless parsing, parser combinators, parsing expression grammars and packrat parsers to model grammars and parsers as objects that can be reconfigured dynamically. Through examples and benchmarks we demonstrate that dynamic grammars are not only flexible but highly practical.

1. INTRODUCTION
It is common practice to define formal grammars using a dedicated specification language which is then transformed into executable form by code generation. Typically these transformation algorithms validate that the grammar is a subset of Context Free Grammars (CFGs), such as LL(k), LR(k), or LALR(k). Then the algorithm optimizes and transforms the grammar into a parser. While this process can give parse-time guarantees and ensure that the parse is unambiguous, the resulting parsers are inherently static. The grammar is hard-coded and cannot be easily changed (at run-time) nor can it be easily composed with other grammars. Numerous researchers have addressed these issues in the past [10, 2, 15, 5]. Earley and SGLR(k) parsers are composeable [6, 16], however the parse results are usually ambiguous and the grammar definition is static and cannot be changed after compilation.

In this paper we present PetitParser, a parser framework that makes it easy to dynamically reuse, compose, transform and extend grammars. We can reflect on the resulting grammars and modify them on-the-fly. We report on the “dynamic” features of this framework and how this is beneficial to grammar development.

The remainder of this paper is structured as follows: Section 2 introduces the PetitParser framework. Section 3 discusses important aspects of a dynamic approach, such as composition, correctness, performance and tool support. Section 4 presents a short overview of the related work, and Section 5 concludes the paper.

2. PETITPARSER
PetitParser takes four existing parser methodologies and combines the best properties of each:

- **Scannerless Parsers** [16] combine lexical and context-free syntax into one grammar. This avoids the common problem of overlapping token sets when grammars are composed. Furthermore language definitions become more concise as there is only one uniform formalism.
- **Parser Combinators** [11] are building blocks for parsers modeled as a graph of composable objects; they are modular and maintainable, and can be changed, recomposed and reflected upon.
- **Parsing Expression Grammars** (PEGs) [8] provide ordered choice. Unlike CFGs, the ordered choice of PEGs always follows the first matching alternative and ignores other alternatives. PEGs are closed under union, intersection, and complement; and can recognize non-context free languages.
- **Packrat Parsers** [7] give linear parse time guarantees and avoid problems with left-recursion in PEGs through memoization [17]. For efficiency reasons PetitParser does not memoize each rule, but only selected ones [1].

PetitParser is implemented in Pharo Smalltalk\(^1\). A detailed description of the API and instructions on how to download the framework and examples can be found in a blog article of the first author\(^2\). Grammars are specified by composing primitive parser objects using a series of overloaded operators forming an internal domain-specific language. For example, the grammar of identifiers is implemented as follows:

```
identifier ::= #letter asParser , (#letter asParser / #digit asParser) star.
```

The expressions #letter asParser and #digit asParser return parsers that accept a single character of the respective character class; the ‘,’ operator combines two parsers to a sequence; the ‘/’ operator combines two parsers to an ordered choice;

\(^1\)http://www.pharo-project.org/  
\(^2\)http://www.lukas-renggli.ch/blog/petitparser-1/
and the ‘star’ operator accepts zero or more instances of another parser. As a result we end up with a graph of connected parser objects that can be used to parse input:

```
identifier parse: 'id12'. "consumes input and returns a default AST"
identifier parse: '123'. "returns parse failure: letter expected at 1"
```

At all times the graph of parser objects remains accessible and mutable. We are able to re-compose, transform and change an existing grammar, as we shall see in upcoming examples.

3. PETITPARSER IN PRACTICE

In the remainder of this paper we use a Smalltalk grammar as the running example. The grammar consists of 242 primitive parser objects that are grouped into 78 productions. One of these productions is the identifier rule we have seen in the previous section. The parser produces a standard Smalltalk AST and passes all 296 unit tests of the original hand-written parser.

3.1 Grammar Specialization

Although complex grammars can be defined using a script, we provide a convenient way to define grammars as part of a class. Each production is implemented using an instance variable and a method returning the grammar of the rule. Productions within the grammar are referred to by accessing the instance variable. This allows us to resolve mutually recursive productions by initializing all slots with a forward reference that is resolved by subsequently calling the production methods [3].

Furthermore, defining grammars in classes enables developers to take advantage of the existing development tools. Additionally, we gain the ability to extend grammars by subclassing, as proposed by Bracha [3]. For example, the Smalltalk grammar is split into various classes inheriting from SmalltalkGrammar, which defines the language grammar only. The subclass SmalltalkParser adds production actions to build the abstract syntax-tree (AST):

```
SmalltalkGrammar>>variable
  ^ identifier

SmalltalkParser>>variable
  ^ super identifier ==>[ :token | VariableNode token: token ]
```

As described in our previous work [14], subclassing gives us the possibility to reuse the same grammar with different tools, such as compiler, editor (syntax highlighting, code completion) and debugger (code evaluation). In a statically typed language this would be more difficult, as each of the grammar subclasses retains its own dedicated data structures.

3.2 Grammar Composition

PetitParser is built around composition: simple parsers are combined into more complex ones. Grammars can arbitrarily be reused and composed. For example, to reuse the grammar for a Smalltalk method declaration we can ask for its production, which is a working parser by itself. Furthermore we can then combine this production with the grammar of another language, such as SQL:

```
```

3.3 Grammar Conflicts

The downside of being able to arbitrarily compose grammars is that this might lead to subtle problems. In the following example we use a language extension that makes it possible to put SQL expressions anywhere in Smalltalk code. The problem in the following example is that the embedded SQL expression could also be parsed as a valid Smalltalk expression:

```
Person>>load
  ^ SELECT * FROM "Person"
```

While the parse is always unambiguous (because of the ordered choice), the result of the above expression depends on the order in which the two languages have been composed. In previous work [13] we argued that this specific language embedding is relatively safe, since SQL is a restrictive language that cannot parse typical Smalltalk expressions. While this works well in practice for this particular example, it might not be feasible for other examples and emerging problems might stay unnoticed.

To avoid this problem we introduced an unordered choice at the merge points. This enforces that exactly one of the two grammar fragments parses. Interestingly the unordered choice ‘|’ is trivial to implement using the semantic negation predicate ‘!’ of PEG parsers (‘not’ in our implementation), however to our knowledge has not been documented.

```
Parser>>| aParser
  "Answer a new parser that either parses the receiver or aParser, fail if both pass (exclusive or)."
  ^ (self not , aParser) / (aParser not , self)
```

The resulting parser enforces that only one of the two choices parses. Contrary to the unordered choice in CFGs our implementation does not work statically, but dynamically at parse-time. Note that the unordered choice should not be used as the default choice operator as it can lead to exponential parse times if both choices are followed on each parse.

On top of that we can define other operators such as a dynamicChoice, which lets the user disambiguate and potentially change the grammar on-the-fly:

```
Parser>>dynamicChoice: aParser
  self | aParser / [ :stream |
  | resolution |
  | resolution := UIManager default
  | chooseFrom: { self name. aParser name }
  | values: { self. aParser }
  | title: 'Resolve ambiguity at ', stream position asString.
  | resolution parseOn: stream | asParser
```

With these extensions, the difference between CFGs and PetitParser is similar to the difference between statically

Footnote 3: In Smalltalk the expression would represent the multiplication of the variables SELECT and FROM, and “Person” would be a comment.
and dynamically typed languages. In both PetitParser and dynamically typed languages, static errors (such as grammar ambiguities or type errors) are detected at run-time only, at the gain of additional flexibility at run-time.

### 3.4 Grammar Transformations

The ability to transform grammars is powerful and goes beyond static extensibility by single inheritance: transformations can be applied on-demand and multiple transformations can be chained.

To highlight code, we can instantiate the basic grammar definition and wrap all parsers that create a token with an action block that highlights the character range in the editor. The backtracking that might occur during the parse is not a problem, because the method highlight:range: overrides the style if set previously. In any case the highlighting purely happens as a side-effect of the parsing.

```plaintext
```

In a similar manner other kinds of common errors can be skipped and the user can be warned while writing the code.

PetitParser also provides a query interface to reflect on parsers: aParser allParsers returns a collection of all parsers in the grammar; aParser firstSet returns the parsers that consume input first in an aParser; aParser followSet returns the parsers that follow aParser; etc. We can combine the same transformation techniques and the reflective facilities to dynamically generate a grammar to answer other questions, such as what could possibly follow at a specific point in a source file:

```plaintext
PPParser>> whatFollows: aString at: anInteger
 | stream |
 stream := aString asPetitStream.
 (self transform: [:parser | parser == > [:node | stream position < anInteger ] ifTrue: [ node ] ifFalse: [ "parser followSet "] ]]
 parseOn: stream.
 #()
```

Furthermore grammars can be searched and transformed using a declarative transformation engine. We are using an unification algorithm on the grammar graph that tries to substitute the patterns in a search expression with actual parsers. If a match is found, the replacement rule is instantiated with the matched parsers and inserted into the grammar. This allows us to concisely specify grammar optimizations.

### 3.5 Tool Support

Figure 1 displays a grammar workspace which provides a rich set of static and dynamic tools that directly work on the object model of PetitParser.

![Figure 1: The PetitParser grammar workspace displaying the currently selected production.](image)

The static tools consist of all elements that work on the specification of the grammar. The source tab enables editing of productions; the graph tab displays the graphical structure of productions; the example tab displays random examples for the selected production, which is useful to spot errors in the grammar definition; the cycles tab lists direct cycles that could cause inefficient grammars; and the first and follow tabs display the respective set of parsers that consume input first and that follow the selected production.

The dynamic tools work on the currently selected production and an input to parse: The parse tab displays and optionally opens an inspector on the resulting AST; the tally tab displays the absolute and relative activation count of each production; the profile tab displays absolute and relative time spent in each production; the progress tab visualizes the progress of the parser through the input — from left-to-right the input string is depicted (whitespaces in white), from top-to-bottom the time (see Figure 2 for an example); and the debugger tab gives the possibility to step forwards and backwards through the input while highlighting the consumed text and the active productions.

### 3.6 Performance

Parser Combinators, PEGs, and even Packrat parsers are often accused of being slow, due to their dynamic nature. Our experience however has shown the contrary: when the grammar is carefully written PetitParser can compete with a highly optimized LL1 table based parser. Efficient grammars can be achieved with automatic grammar optimization transformations (Section 3.4) and the help of the static and dynamic tools (Section 3.5).

In Table 1 we list the average throughput of different Smalltalk parsers producing an identical AST. The hand-written recursive descent parser is a clear winner, being almost 5
times as fast as the other two parsers. We expected the LALR parser [4] to perform better, given the sophisticated optimization algorithms implemented in this compiler-compiler. Profiling the parsers reveals that the LALR parser spends most of its time looking up, decoding and dispatching values from its tables. PetitParser on the other hand shows a deep nesting of message sends, something a dynamic language like Smalltalk can do very efficiently.

4. RELATED WORK

OMeta [18] is an object-oriented pattern matcher based on a variant of PEGs. OMeta uses itself to transform grammar specifications to host language code. Rats! [9] is a packrat parser framework that provides a sophisticated infrastructure to transform and optimize grammars using the visitor design pattern. Both frameworks support grammar composition, but due to their code generation make it impossible to change the grammar after compilation.

Various other object-oriented frameworks for parser combinators have been proposed: JParsec, Scala Parser Combinators [12], and Newspeak [3]. All implementations use the host language to build an object model of parser objects. Extensibility is achieved through subclassing or mixing mechanisms of the respective host languages. Although we expect that grammar transformations are possible on these models, we are unaware if this has actually been done in practice.

Composing and reusing table based parsers is an ongoing research topic [10, 2, 5]. All approaches have limitations in composability and can be described as difficult at best. Grammar changes require an expensive recompilation of the new grammar. Schwerdfeger et al. [15] propose a solution to efficiently compose table based grammars at specific extension points.

5. CONCLUSION

Our work on language embedding made us wonder why grammar definitions in dynamic languages are still mostly hard-coded statically. In this paper we advocate the use of dynamic grammars. We have presented several examples and some basic performance analysis that show the benefit of having grammars be accessible and changeable at run-time.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Bringing Models Closer to Code” (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010). We also like to thank Jorge Russia and the anonymous reviewers for their feedback on this paper.

6. REFERENCES