Enabling Software Analysis using PetitParser and Moose

Michael Rüfenacht
Software Composition Group
University of Bern, Switzerland
m.ruefenacht@students.unibe.ch
http://scg.unibe.ch

***Abstract*—Static software analysis is an important process in software quality assurance. Building suitable tools e.g. parsers and the processing (or even derivation) of underlying grammars can be a tedious task. Creating parsers by hand involves a lot of repetitive work and is error prone, hence it is recommendable to rely on suitable frameworks and systems.**

This paper describes a methodology to enable source code analysis of a programming language using *PetitParser*[1] and *Moose*[2]. We will use the *PetitParser* framework to implement a parser and create an adequate *Intermediate Representation* (IR). Furthermore, we will apply suitable transformations to create a meta model which can be queried and analyzed using *Moose*.

I. Introduction

Often, software analysis tools have a pretty static nature, concerning both the source code parsing and analysis. Parsers and parser generators are built upon static definitions and generate static parsers. Hence these systems provide limited flexibility when it comes down to changes and composite/heterogeneous systems. The integration of *PetitParser* into the *Moose* suite — both written in Pharo Smalltalk — provides a highly flexible and generic environment for reverse- and re-engineering purposes.

On one hand we have *PetitParser*, which allows a concise way to transpose a grammar into a scannerless parser using an internal domain specific language and solves a number of common problems arising when parsing Context Free Grammars (CFGs) such as ambiguity. Also, it provides a visualization panel, which adds evaluation capabilities to the underlying parser to establish significant improvements on the implementation.

On the other hand we have Moose, a software analysis suite which consumes meta representations of data such as parsed source code. *Moose* enables browsing, reasoning on and visualization of a given model, depending on any particular need or metric.

Nevertheless, during the implementation, several obstacles and caveats inevitably arise and it is advisable to have some simple patterns in mind when creating the various components. This paper combines the knowledge of several resources and practical insights to present a initiating guideline to the usage and especially integration of the mentioned tools.

Outline. The paper covers an introduction to *PetitParser* in Section II, introducing the different types of parsers and tools of the framework. Section III describes how to set up the environment and Section IV overviews the architectural approach taken when implementing an extension to *Moose*. Section V subsequently describes the required components, provides insights on their implementation and testing. Section VI lists a number of problems which possibly arise and corresponding solutions. Section VII gives an overview over work similar to this paper before Section VIII concludes this paper.

II. PetitParser in a nutshell

The PetitParser framework enables the programmatic composition of grammars. It consists of several parser types and useful factory methods for a readable and convenient access and composition of parsers. To ensure the desired flexibility and expressiveness, it combines the advantages of four parsing techniques:

- Scannerless Parsing [1] *avoids the additional lexical analysis and simplifies the syntax into one formalism*
- Parser Combinators [2] *arranges the parsers as modular components into a graph to enable combinatory parsing*
- Parsing Expression Grammars (PEG) [3] *provide ordered choice (always follow first matching alternative)*
- Packrat Parsing [4] *ensures linear parse time and avoids left recursion problems of PEGs via memoization and infinite lookahead*

[1] http://scg.unibe.ch/research/helvetia/petitparser
[2] http://www.moosetechnology.org/

To get a deeper insight and access to the underlying references have a look at the publications of Lukas Renggli [5], [6], the author of PetitParser.

A. Terminal Parsers

Terminal parsers are the fundamental components of a parser generated with *PetitParser*. Terminal symbols of a language can be modelled from a *literal object* (i.e. characters and numbers) or *literal sequences* (i.e. strings) using the `asParser` selector. In addition, *PetitParser* enables the creation of predefined ranges sending `asParser` to symbols which correspond to a factory method[3] (such as the `#letter` or `#digit` predicates). Table I presents a list of the predefined literal parsers, whereas Listing 1 demonstrates how they can be used.

TABLE I: Overview over terminal parsers

Parser	Matches
$a asParser	'a'
'a' asParser	'a'
'abc' asParser	'abc'
#any asParser	. *(any character)*
#lowercase asParser	[a-z] *(isLowercase)*
#uppercase asParser	[A-Z] *(isUppercase)*
#letter asParser	[a-zA-Z] *(isLetter)*
#digit asParser	[0-9] *(isDigit)*
#hex asParser	[0-9a-fA-F]
#word asParser	#letter | #digit
#space asParser	*Character isSeparator*
#tab asParser	*Character tab*
#blank asParser	*#space | #tab*
#newline asParser	*Character cr | Character lf*
#punctuation asParser	*.,"'?!;:#$%&()*+-/<>=@[]\^_{}|~*

Overview over literal parsers and a description of their matching equivalent (*isXY* emphasize the predicate).

```
1 booleanTrue := 'true' asParser
2 booleanTrue parse: 'true'
```

Listing 1: A boolean literal parser

B. Combination and Repetition

To combine the created terminal parsers, *PetitParser* offers a concise and convenient syntax (analogous to PEG). Combining parsers programmatically will result in expressive code that can be read like a grammar. Table II shows parser combinations of *PetitParser* and Table III lists repeating parsers. In Listing 2, *booleanTrue* - previously introduced in Listing 1 - is combined with a *booleanFalse* parser.

```
1 booleanTrue := 'true' asParser.
2 booleanFalse := 'false' asParser.
3 boolean := booleanTrue / booleanFalse.
```

Listing 2: A choice composition

TABLE II: Parser Combinations

Parser	Equals
p , q	p q *(sequence)*
p / q	p / q *(ordered choice)*
p optional	p / ϵ
p and	&p *(non consuming lookahead)*
p not	!p *(non consuming lookahead)*
p negate	!p , any *(consuming)*
p | q	(!p q) / (q! p) *(XOR, unordered choice)*
p caseInsensitive	
p end	*succeeds only if p succeeds directly before the end of the stream*

Combinations of parsers where p, q are arbitrary parsers

TABLE III: Repeating Parsers

Parser	Equals
p times: n	p^n
p min: n max: m	$p^{[n \ldots m]}$
p star	p * *(zero or more times)*
p +	p + *(one or more times)*
p separatedBy: q	p (q p)*
p delimitedBy: q	p (q p)* (q / ϵ)

Repeating parsers where p, q are arbitrary parsers.

C. Action parsers

It is often necessary to transform the result of the parsing to get a more suitable representation e.g. by consuming pre- and succeeding stream positions (i.e. characters of the input string). A parser can be converted into an `PPActionParser` using the `==> aBlock` selector, where the block receives the parsing result as an argument and returns the desired representation.
The *boolean* parser (Listing 2), can be enhanced by the ability to consume whitespace around the literal parser and transform the result from a string to its actual boolean value, as shown in Listing 3. Table IV provides an overview over the available action parsers.

[3]Defined in the `PPPredicateObjectParser class` http://de.slideshare.net/renggli/mastering-grammars-with-petitparser

```
1  booleanTrue := 'true' asParser
2                         ==> [:t | true].
3  booleanFalse := 'false' asParser
4                         ==> [:t | false].
5  boolean := (booleanTrue / booleanFalse) trim.
```

Listing 3: The action parser consumes whitespace and returns boolean values

TABLE IV: Action Parsers

Parser	Action
p token	*creates a token out of the result*
p trim	*consumes whitespace before and after p*
p trim: aTrimmer	*consumes positions where aTrimmer succeeds before and after p*
p ==> aBlock	*applies the block as a transformation on the result (aBlock receives the result as parameter)*
p flatten	*flattens the result to a string*
p foldLeft: aBlock	*folds the result into a block (also see `foldRight`)*

Action parsers transform the parsing result.

D. Expression parsers

Creating parsers that can handle complex expressions (or even more: create well defined grammars which e.g. include expressions) can be difficult. Different operator types, operator precedence and associativity have to be handled properly. *PetitParser* comes with `PPExpressionParser`, able to overcome such difficulties. `PPExpressionParser` creates precedence by grouping operators, has the capabilities to define pre-, post- and infixes and enables direct result transformation. Listing 4 shows an expression parser which handles and evaluates additions and subtractions. The `left:` and `right:` selectors (line 9 and 11) enable the desired left- or right-associativity. For instance evaluating `expr parse: '100 + 10 - 20++'` results in `89`.

E. Composite Parser

The `PPCompositeParser` is the base class for the creation of grammars and parsers in an object-oriented way and encapsulates combinations of parsers. Using a composite-parser introduces all benefits of object-oriented programming (such as modularity and re-usability) to the adaption of a grammar. Combining composite parsers even allows the processing of heterogeneous systems by combining whole grammars. Section IV and Section V describe the usage of a composite-parser by taking the example of a contrived programming language.

```
1  expr ::= PPExpressionParser new
2    term:
3      (#digit asParser plus flatten trim)
4      ==> [:num | num asNumber ];
5    group: [:g |
6      g   prefix: '--' asParser trim
7          do: [:o :a | a - 1 ].
8      g   postfix: '++' asParser trim
9          do: [:a :o | a + 1 ]];
10   group: [:g |
11     g   left: $+ asParser trim
12         do: [:a :op :b | a + b ].
13     g   left: $- asParser trim
14           do: [:a :op :b | a - b ]].
```

Listing 4: Overview over the expression parser. Groups create precedence in descending order.

F. Debugging and Tryout

PetitParser contains a browser that allows modification, inspection and visualization of implemented grammars and parsers (subclasses of `PPCompositeParser`). A `PPBrowser` can be opened via *World→Tools→PetitParser* or invoking `PPBrowser open`. The browser provides a number of features to work with composite grammars such as

- source code view and editing
- graph and map visualizations
- first and follow set computation
- parsing and analyzing of sample code (debugging, profiling, progress visualization)

Furthermore the browser is a good entry point to try out code and detect erroneous behavior to refine and improve tests accordingly.

III. Setup

The *Moose* suite is distributed as a Pharo image and can be downloaded from http://www.moosetechnology.org/download. Besides a standard *Moose* distribution the image also contains the *PetitParser* framework.

IV. Approach

To exploit analysis capabilities of *Moose* for a programming language, we are using the aforementioned `PPCompositeParser`, visitors and node-transformation. The creation of a *PetitParser* is done *bottom-up* which means to start by creating terminal parsers and subsequently compose them to productions. Unlike the creation workflow, the parsing is executed *top-down*. This is important to keep in mind since it introduces constraints on the modeled grammar [7]. It is recommendable to monitor the correctness of the grammar by following a

test-driven methodology since even minor changes may break the parsing.

To make the resulting parse-tree processable by *Moose* requires transformation. Elements of the resulting *Abstract Syntax Tree* (AST) are converted into *Intermediate Representations* (IRs) using visitors. IRs provide convenient access to the properties of the underlying parse and expose entry points for visitors that apply reifications on the created tree structure. Target of the transformations is to create a final IR consisting of FAMIX [8] nodes, an unified meta model of object-oriented source code. Figure 1 depicts an abstraction of the workflow and its components.

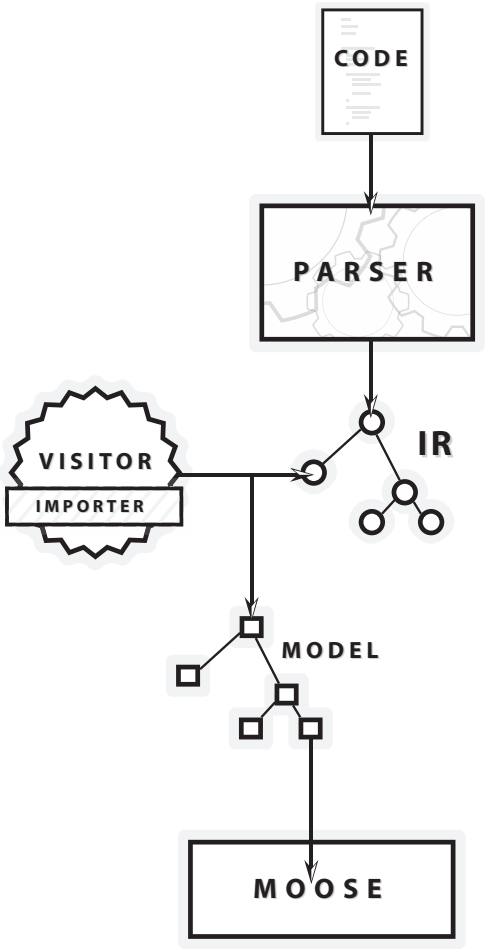

Fig. 1: An overview of the approached workflow.

V. Implementation of SmallPHP

The following examples are based on a contrived subset of *PHP*[4] inspired by the ideas of *MiniJava*[5]. A possible grammar for *SmallPHP* is appended in Appendix A.

A concrete implementation following the discussed approach, mainly needs the components depicted in Figure 2.

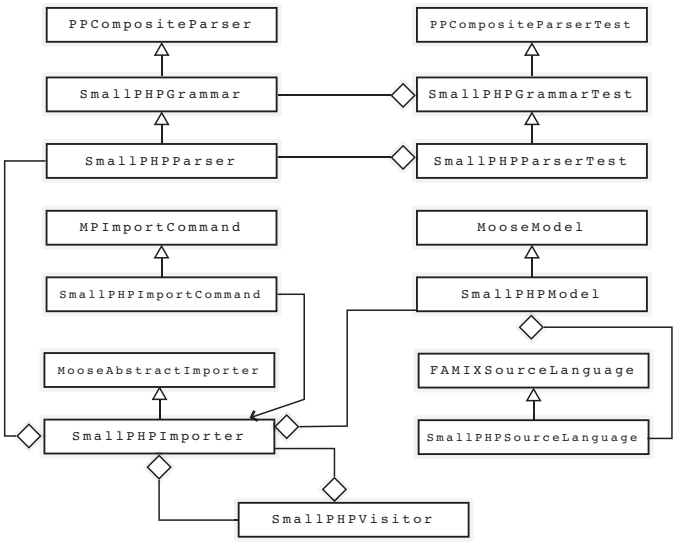

Fig. 2: An overview of the components and their relations.

A. The Grammar

The core grammar is a subclass of the `PPCompositeParser` and defines all the productions and literals of the grammar.

```
1 PPCompositeParser subclass: #SmallPHPGrammar
2   "instance variables contain all elements"
3   instanceVariableNames: '...'
4   classVariableNames: ''
5   category: 'SmallPHP-Core'
```

Listing 5: Defining the grammar

Every element of the grammar corresponds to an instance variable[6]. The `SmallPHPGrammar` object has to provide equally named accessor-methods returning the dedicated parsers for each instance variable. `PPCompositeParser` will set up all the defined parsers on initialization as `PPDelegateParser` to avoid problems with mutually recursive definitions (see `PPCompositeParser>>initializeStartingAt: aSymbol`). Leaving out an accessor-method for any production will cause the parser to fail.

Furthermore it is necessary to implement the `start` method which is the entry point for parsing (often

[4]http://php.net/

[5]http://www.cambridge.org/us/features/052182060X/

[6]As long as a production is not accessed directly via an accessor-method which would make the code harder to read and less concise

referred to as `goal`) which is shown in Listing 6. The declaration is concise and directly corresponds to the definition in Appendix A. To get a feeling on how the productions look, please consider Listing 7 where a possible implementation of a namespace-statement production is listed. Listing 7 also shows why the parsers are created on initialization: when `namespaceStatement` gets called, the instance variable `identifier` would be undefined i.e. **`nil`**.

```
SmallPHPGrammar >> start
  ↑ openDelim ,
    namespaceStatement optional ,
    (classDeclaration / statement) star,
    closeDelim end
```

Listing 6: The start method: entry point for the parsing.

```
SmallPHPGrammar >> identifier
  |us|
  "underscore"
  us := $_ asParser.
  ↑ ((#letter asParser / us),
      (#word asParser / us) star trim

SmallPHPGrammar >> namespaceStatement
  |separator|
  separator := $\ asParser.
  ↑ 'namespace' asParser token trim,
      separator trim optional,
      identifier separatedBy: separator,
      $; asParser trim
```

Listing 7: A namespace statement implementation

B. Testing the Grammar

As stated, it is important to test the implemented productions and creating a test suite is straightforward as shown in Listing 8. To hook in a custom parser into the tests (the grammar in this case), it is necessary to override `PPCompositeParserTest >> parserClass`. The test suite now allows a verification of the defined rules by checking if a production holds on a code segment (Listing 10) or explicitly fails (Listing 11). The explicit failure checking of rules is extremely useful to create test cases that indicate possible ambiguities in the grammar and ensure the chosen solution works as expected.

```
PPCompositeParserTest subclass:
                  #SmallPHPGrammarTest
  category: 'SmallPHP-Core-Tests'
```

Listing 8: The testsuite

```
SmallPHPGrammarTest >> parserClass
  ↑ SmallPHPGrammar
```

Listing 9: Hooking in a custom parser

```
SmallPHPGrammarTest >> testBooleanTrue
  self parse: 'true' rule: #booleanTrue
```

Listing 10: Testing grammar rules

```
SmallPHPGrammarTest >> testBooleanTrueFails
  self fail: 'false' rule: #booleanTrue
```

Listing 11: Ensure failure of rules

As soon as all the terminal parsers are implemented the code visually and semantically corresponds to convenient syntaxes for grammar definitions (c.f. Listing 2) and can be easily tested (Listing 12)[7].

```
SmallPHPGrammarTest >> testBoolean
  self parse: 'false' rule: #boolean.
  self parse: 'true' rule: #boolean.
  self assert: result isTrue.
```

Listing 12: Testing the #boolean rule

C. Intermediate Representation

As elaborated in Figure 1, the parser will generate an *Intermediate Representation* (IR) from the initial AST. Before implementing the parser we have to prepare the nodes of the IR. To make the parse tree traversable, all of its nodes are visitable by a visitor and hence implement an `accept:` method. Basically, these nodes are value objects. Since Smalltalk is a dynamic typed language, the visitor methods cannot be distinguished by overloading and have to be specified explicitly for every node type. Listing 13 presents a TypeNode's `accept:` method which performs a double dispatch.

```
SmallPHPNamedNode subclass: #SmallPHPTypeNode
  instanceVariableNames: 'supertype variables
    methods'
  category: 'SmallPHP-Nodes'

SmallPHPTypeNode >> accept: aVisitor
  ↑ aVisitor visitSmallPHPTypeNode: self
```

Listing 13: Excertps from a SmallPHPTypeNode

D. The Parser

From the grammar, we derive a parser. The parser is a subclass of the grammar and introduces further processing of the results. In terms of parsing, the defined grammar class does lexing and parsing, while the parser we are about to introduce, does the transformations. To improve the initial parsing and create the IR, we overwrite the

[7]The example combines the two cases true and false into one test. It is highly recommended to split them up to get a fine grained insight into possible error sources.

relevant methods of the grammar. The parsers specified and returned by the grammar rules are transformed into action parsers as listed in Table IV, which create the the IR. Listing 14 contains an example of a transformation, taking the parser from the SmallPHPGrammar and adding a transforming block. It is useful to make use of the double-dispatch pattern if the type of the member node is not directly given by the context.

```
SmallPHPGrammar subclass: #SmallPHPParser
  category: 'SmallPHP-Core'

SmallPHPParser >> classDeclaration
  ↑ super classDeclaration ==> [ :token |
    | node |
    node := SmallPHPTypeNode new.
    node name: token second.
    node supertype: token third.
    node stub: false.

    token fourth ifNotNilDo: [:body |
      body do: [:member |
        "double dispatch"
        member addToParent: node.
      ].
    ].
    node
  ]
```

Listing 14: Defining the parser and a transformation that creates an IR node out of the AST

E. Testing the parser

Creating a test suite for the parser is as simple as subclassing the grammar testing suite. Nevertheless the testing slightly differs from the tests done for the initial grammar. Instead of directly invoking rules and simply test for success, the parser tests overwrite and invoke the tests from the grammar and check if the created nodes correctly map the parsed code. The result of the parsing is available via the `result` property of `PPCompositeParser`.

```
SmallPHPGrammarTest subclass:
    #SmallPHPParserTest
  category: 'SmallPHP-Core-Tests'

SmallPHPParserTest >> testClassDeclaration
  "stores the parsing result"
  super testClassDeclaration.

  self  assert:
        (result isKindOf: SmallPHPTypeNode).
  self  assert: result name
        equals: 'DoubleIterator'.
  self  assert: result parent name
        equals: 'Iterator'.
  self  assert: result variables size
        equals: 2.
  self  assert:
        (result hasVariable: 'storage').
  ...
```

Listing 15: Defining the parser test suite and testing nodes

F. Visitor & Importer

To analyze and process the nodes generated by the parser, we use visitors. The usage of visitors provides a flexible and interchangeable way of applying actions on complex data structures. In our case the visitor only guides the importer through the syntax tree and returns the result as presented in Listing 16. The importer itself is a subclass of `MooseAbstractImporter` which allows the importers to share error logging and an importer context. `SmallPHPImporter` analyzes the nodes and transforms the IR into *Famix*[8] nodes which are processable by Moose shown in Listing 17 and 18. The importer maintains according symbol tables to avoid type duplications.
The composition of visitor, importer and model does not follow a strict pattern and is applied differently in other implementations (there does not seem to be one best way). Relying on the visitor in the importer as in Listing 18 may look cumbersome and redundant, since the results of the parsing are unambiguous. Nevertheless it ensures an additional level of abstraction and avoids type checking by providing the necessary context.

```
Object subclass: #SmallPHPVisitor
  instanceVariableNames: 'importer'
  category: 'SmallPHP-Nodes'

SmallPHPVisitor
      >> visitSmallPHPTypeNode: aTypeNode
  ↑ self importer ensureAType: aTypeNode
```

Listing 16: Excerpt from the visitor implementation

```
MooseAbstractImporter subclass:
    #SmallPHPImporter
  instanceVariableNames: 'visitor parser ...'
  category: 'SmallPHP-Moose'

SmallPHPImporter >> ensureAType: aTypeNode
  | name |
  name := aTypeNode name.
  "types are stored in a dictionary,
  identified by their name"

  (self hasTypeFor: name)
    ifTrue: [ ↑ self types at: name ]
    ifFalse: [ ↑ self createType: aTypeNode ].
```

Listing 17: Excerpts from the importer code including the type ensuring from Listing 16 (line 8)

[8] *Famix* is a meta-model shipped with *Moose* http://www.moosetechnology.org/docs/famix

```
SmallPHPImporter >> createType: aTypeNode
  | node identifier |
  identifier := aTypeNode name.
  node := FAMIXClass new.
  node name: identifier.

  self types at: identifier put: node.

  "attach the visitor to variables..."
  aSmallPHPTypeNode variables do:
        [ :var |
          (var accept: visitor)
          parentType: node ].

  "and methods"
  aSmallPHPTypeNode methods do:
        [ :var |
          (var accept: visitor)
          parentType: node ].

  "ensure inheritance"
  aSmallPHPTypeNode parent
    ifNotNilDo: [ :parent |
      | inheritance superclass |
      superclass := parent accept: visitor.
      inheritance := FAMIXInheritance new
        superclass: superclass;
        subclass: node.
      node addSuperInheritance: inheritance.
      superclass addSubInheritance:
    inheritance ].
  ↑ node
```

Listing 18: Simplified FAMIXType creation

G. The model

Before being imported, all the nodes created need to be stored in a container: a `MooseModel`. In our case the moose model only gets enhanced by helper methods, to tie the importer, the model itself and the source code together. These helper methods are also useful to add additional functionality like UI integration during the importing task. Listing 19 shows the relevant methods of the Model for importing using the importer's `import: aStream` method as shown in Listing 20. The model sets itself as the target model for the importer which parses the stream and populates the model with the resulting nodes as presented in Listing 20. Other than the example, all elements of the AST have to be added to the model. Otherwise the analysis may fail.

```
MooseModel subclass: #SmallPHPMooseModel

SmallPHPMooseModel >> importSmallPHPFile
  | stream |
  stream := UITheme builder
    fileOpen: 'Import model from .php file.'
    extensions: #('php').
  stream isNil ifFalse: [
    self name: (FileDirectory baseNameFor:
    stream localName).
    self importFromPHPStream: stream.
    stream close].

SmallPHPMooseModel >> importFromPHPStream:
    aStream
  | importer |
  importer := SmallPHPImporter targetModel:
    self.
  importer import: aStream.
```

Listing 19: The import helpers of the SmallPHPMooseModel

```
SmallPHPImporter >> import: aStream
  (parser parse: aStream) do:
    [:item |
      |type|
      type := item accept: visitor.
      self targetModel silentlyAdd: type.
      self targetModel silentlyAddAll: type
    methods.
      "..."
    ].
```

Listing 20: Simplified model population in `SmallPHPImporter >> import: aStream`, adding type nodes and their methods

H. Wrapping it up

Having a working parser, visitor, importer and model, the setup is ready to import code into the `MoosePanel` which visualizes the meta model. One can open the panel under *World→Moose→MoosePanel* in a *Moose* image, or by invoking `MoosePanel open`.
The last missing component is an import command to install the model. An import command is a subclass of `MPImportCommand` and overwrites its `label` and `execute` methods. There is no need to manually integrate the command into the panel's user interface since the panel searches for all (kinds of) `MoosePanelCommands` and integrates them into the toolbar. The implementation of such an importer command is presented in Listing 21. The installation adds the model to the cache of *Moose* and hence makes it available in the `MoosePanel` which provides the interface for browsing and visualization (for more information have a look at the MooseBook[9]).

```
MPImportCommand subclass:
    #SmallPHPImportCommand

SmallPHPImportCommand >> label
  ↑ 'Import SmallPHP code from a .php file'

SmallPHPImportCommand >> execute
  | model |
```

[9] http://www.themoosebook.org/book

```
  model := SmallPHPMooseModel new.
  model importSmallPHPFile.

  model size > 0 ifTrue: [
      model install.
      self addModel: model.
    ].
```

Listing 21: Implementation of an importer-command

VI. Troubleshooting

The following points briefly describe some difficulties arising during the implementation of a parser and the integration into Moose.

A. False positives in tests

The way we implemented the grammar-tests only ensures a successful parse but does not fully ensure its correctness (e.g. the resulting AST). Sometimes it is important to validate the structure of the parsing result as an indicator that productions did not consume an incorrect amount of input (e.g. due to an imprecisely defined grammar). There are two possible ways to ensure a correct parsing:

1) Analyze the result variable as we did in Listing 15 for the parser.
2) `PPCompositeParserTest` provides a method to compare the result of the parsing to the expected value: `assert: aString is: anObject` and `assert: aParser parse: aString to: anObject`. Unfortunately there is no direct way to control the applied production but `PPCompositeParserTest` provides the necessary tools to achieve the desired effect. Listing 22 shows how to validate the test result for a specific grammar rule.

```
SmallPHPGrammarTest >> testRulename
  |expected|
  expected := {"..."}
  self
    assert:
      (self parserInstanceFor: #ruleName) end
    parse: 'someCode'
    to: expected.
```

Listing 22: Test to compare the parsing result of a specific rule (i.e. #ruleName)

B. Ignored Names

As mentioned in subsection V-A, `PPParser` instances initialize all their instance variables as unresolved parsers. As a side effect it is not possible to have additional instance variables by default. `PPParser class` provides an *ignored names* method which returns a collection of variable names to ignore. We can extend the default collection as shown in Listing 23 to allow additional instance variables.

```
SmallPHPGrammar class >> ignoredNames
  ↑ super ignoredNames , #('varname1' '
    varname2')
```

Listing 23: Ignoring instance variables

C. Expressions

The presented approach using a `PPExpressionParser` provides a number of benefits, such as the precedence handling. Nevertheless the usage of the expression parser involves a loss of context: a subclass will not be able to determine which expression was successful.
To exploit the benefits of using the `PPExpressionParser` without losing context, one can introduce additional helper methods that process the result of the parsing. These helpers can be overwritten by the derived parser and allow hooking up operations into the context of an expression. Listing 24 lists a slightly modified version of an expression parser (c.f. Listing 4) that illustrates the benefits of the helper methods.

```
group: [:g |
  g prefix: '--' asParser trim
    do: [:o :a | self decrement: a ].
  g postfix: '++' asParser trim
    do: [:a :o | self increment: a ]];
  "other operators or groups"

Grammar >> decrement: aResult
  ↑ aResult

Parser >> decrement: aResult
  ↑ aResult - 1
```

Listing 24: How to use helpers to gain context in an expression parser

D. Recursion

The *PetitParser* framework is able to handle mutual recursion by relying on `PPDelegateParser`. Delegating parsers are able to be specified after their initialization which is exactly the approach `PPCompositeParser` takes when setting up its rules (as elaborated in Section V).
PetitParser implements a number of features of Packrat Parsing such as memoization [4] and is therefore able to handle (indirect) left recursion [9], [10] if the parser in

the cycle is memoized (c.f. the test `PPComposedTest >> testLeftRecursion`). Nevertheless it is recommended to rewrite the grammar if it includes left recursion.

E. Error detection & error recovery

We did not actually care about possibly arising errors, be it caused by the implementation or the analyzed source code. The PetitParser framework introduces a `isPetitFailure` method which only evaluates to **`true`** if called on a `PPFailure` object which is returned as a parsing result on failure. The failure object also provides an error message and the position in the underlying stream.

The implementation of an error recovery strategy for a scannerless parser [11] goes beyond the scope of this guide. A brief proposal could be to skip erroneous passages by negating the next expected token as illustrated by Figure 3. The <MethodHeaderForward> rule negates the occurrence of a method header, forwards the underlying stream to the next valid method node and allows the parser to proceed the parsing. This could be done using a `PPPluggableParser` - created by sending the `asParser` selector to a block - a parser that allows to do the parsing manually. Listing 25 presents a snippet for a parsing rule that applies the parsing of its superclass to the stream. Afterwards it checks if the results array contains a failure and if true it applies the negated method header to the stream. It is clear that this strategy is far from being complete since it assumes that there is another method following the actual context.

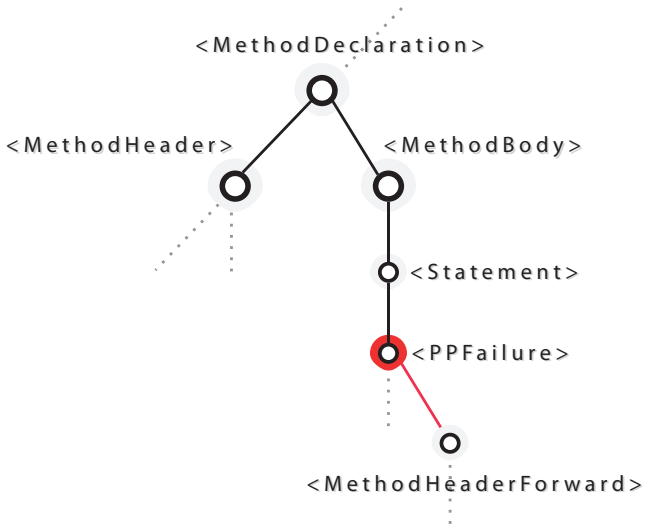

Fig. 3: An illustration on how the parse could be bypassed.

```
SmallPHPParser >> methodBody
  ↑ [:stream |
    | results |
    statements := super methodBody parseOn:
    aStream.
    statements do: [:statement |
      statement isPetitFailure ifTrue: [
        ↑ self  methodHeader
            negate star parseOn: aStream
      ].
    ].
    ↑ statements
  ] asParser
```

Listing 25: Statement rule illustrating an error recovery approach by forwarding the stream to the next method header

VII. Related Work

There seems to be no guide that combines the building of the parser and its integration into *Moose*. The MooseBook[9] offers separate chapters on both, *PetitParser* and *Moose* and is part of the official *Moose* documentation[10]. *PetitParser* is described in the publications of its author, Lukas Renggli [5], [6] and in articles from his website[11] [12]. *PetitParser* will have its own dedicated chapter in the upcoming 'Pharo by Example 2' book[13] (a draft of the chapter can be found on the website).

VIII. Conclusion

The combination of *PetitParser* and *Moose* enables the creation of modular and highly flexible reverse and re-engineering tools for any kind of language. This paper briefly describes the steps to take when implementing the necessary components to extend *Moose*'s analysis capabilities to support another programming language. We walked through the necessary parsers, importer and visitor to get a feeling on how components work together and are integrated into the *Moose* environment. We did not cover any sufficient solutions for error handling and did not further elaborate any implementation details. To get a deeper insight into concrete implementations, one can have a look at the `PetitSmalltalk` or the `PetitSQLite` packages (both included in the *Moose* image). Furthermore we did not elaborate the possibilities of splitting multiple transformation tasks or semantic analysis into different visitors.

Acknowledgments

Thanks to Fabrizio Perin and Oscar Nierstrasz for supporting and pushing me. Also, I thank Jan Kurš and Marc Wiedmer for their reviews.

[10] http://www.moosetechnology.org/docs
[11] http://www.lukas-renggli.ch/blog/petitparser-1/
[12] http://www.lukas-renggli.ch/blog/petitparser-2/
[13] http://rmod.lille.inria.fr/pbe2/

References

[1] E. Visser, "Scannerless generalized-LR parsing," Programming Research Group, University of Amsterdam, Tech. Rep. P9707, Jul. 1997. [Online]. Available: http://www.cs.uu.nl/people/visser/ftp/P9707.ps.gz

[2] G. Hutton, "Higher-order functions for parsing," *Journal of Functional Programming*, vol. 2, no. 3, pp. 323–343, 1992.

[3] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '04. New York, NY, USA: ACM, 2004, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/964001.964011

[4] ——, "Packrat parsing: simple, powerful, lazy, linear time, functional pearl," in *ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, vol. 37/9. New York, NY, USA: ACM, 2002, pp. 36–47. [Online]. Available: http://pdos.csail.mit.edu/~baford/packrat/icfp02/packrat-icfp02.pdf

[5] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz, "Practical dynamic grammars for dynamic languages," in *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, Jun. 2010. [Online]. Available: http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf

[6] L. Renggli, "Dynamic language embedding with homogeneous tool support," PhD thesis, University of Bern, Oct. 2010. [Online]. Available: http://scg.unibe.ch/archive/phd/renggli-phd.pdf

[7] D. Rosenkrantz and R. Stearns, "Properties of deterministic top-down grammars," *Information and Control*, vol. 17, no. 3, pp. 226 – 256, 1970. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019995870904468

[8] S. Tichelaar, "Modeling object-oriented software for reverse engineering and refactoring," Ph.D. dissertation, University of Bern, Dec. 2001. [Online]. Available: http://scg.unibe.ch/archive/phd/tichelaar-phd.pdf

[9] A. Warth, J. R. Douglass, and T. Millstein, "Packrat Parsers Can Support Left Recursion," in *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, ser. PEPM '08. New York, NY, USA: ACM, 2008, pp. 103–110. [Online]. Available: http://doi.acm.org/10.1145/1328408.1328424

[10] A. Warth, "Experimenting with Programming Languages," Ph.D. dissertation, Los Angeles, CA, USA, 2009, aAI3351770. [Online]. Available: http://www.vpri.org/pdf/tr2008003_experimenting.pdf

[11] R. Valkering, "Syntax Error Handling in Scannerless Generalized LR Parsers," Master's thesis, 2007. [Online]. Available: http://homepages.cwi.nl/~paulk/thesesMasterSoftwareEngineering/2007/RonValkering.pdf

Appendix A
SmallPHP Grammar

```
open_delim  ← '<?php' / '<?'
identifier  ← (letter / '_') (digit / letter / '_')*

variable    ← '$' identifier
this        ← '$' 'this'

ns_statement  ← 'namespace' ns_path ';'
ns_path       ← (identifier ns_segment*)
              / ns_segment +
ns_segment  ← ns_separator dentifier
ns_separator  ← '\\'

superclass  ← 'extends' identifier
visibility  ← 'public' / 'private' / 'protected'
class_decl  ← 'class' identifier superclass? class_body
class_body  ← '{' (class_var / method_decl)* '}'
class_var ← visibility variable ';'
class_method  ← visibility 'function' identifier '(' class_method_parameters? ')' class_method_body
class_method_parameters ← variable (',' variable)*
class_method_body ← '{' statement* '}'
statement   ← ('{' statement* '}') /
          ('if' '(' expression ')' statement 'else' statement) /
          ('while' '(' expression ')' statement) /
          ('echo' expression ';') /
          ('return' expression ';') /
          (expression '=' expression ';') /
          (expression ';')

expression  ← (expression ('&&' / '<' / '>' / '+' / '-' / '*' / '==') expression) /
          (expression '[' expression ']') /
          (expression '->' identifier '(' expression (',' expression)*)? ')') /
          (expression '->' identifier) /
          integer /
          'true' /
          'false' /
          'null' /
          this /
          variable /
          ('array' '(' ')') /
          ('new' identifier '(' ')') /
          ('!' expression) /
          ('(' expression ')')

goal      ← open_delim s_statement (class_decl / statement)* close_delim
close_delim   ← '?>'
```