Chapter 1

PetitParser

with the participation of:
Lukas Renggli *(renggli@gmail.com)*

Building parsers to analyse data is a common tasks in software development. In this chapter we present a powerful parser frameworks named PetitParser. And contrary to its name, PetitParser is a really powerful parsing framework combining several technologies (scannerless parsers, parser combinators..). PetitParser is written by Lukas Renggli as part of his work on the Helvetia system but it can be used as a standalone tool.

Building parsers to analyse data is a common tasks in software development. In this chapter we present a powerful parser frameworks named PetitParser. And contrary to its name, PetitParser is a really powerful parsing framework combining several technologies (scannerless parsers, parser combinators..). PetitParser is written by Lukas Renggli as part of his work on the Helvetia system but it can be used as a standalone tool.

1.1 Writing Parsers with PetitParser

PetitParser is a parsing framework different to many other popular parser generators. For example, it is not table based such as SmaCC or ANTLR. Instead it uses a unique combination of four alternative parser methodologies: scannerless parsers, parser combinators, parsing expression grammars and packrat parsers. As such PetitParser is more powerful in what it can parse and it arguably fits better the dynamic nature of Smalltalk. Let's have a quick look at these four parser methodologies:

Scannerless Parsers combine what is usually done by two independent tools (scanner and parser) into one. This makes writing a grammar much simpler and avoids common problems when grammars are composed.

Parser Combinators are building blocks for parsers modeled as a graph of composable objects; they are modular and maintainable, and can be changed, recomposed, transformed and reflected upon.

Parsing Expression Grammars (PEGs) provide ordered choice. Unlike in parser combinators, the ordered choice of PEGs always follows the first matching alternative and ignores other alternatives. Valid input always results in exactly one parse-tree, the result of a parse is never ambiguous.

Packrat Parsers give linear parse time guarantees and avoid common problems with left-recursion in PEGs.

Loading PetitParser

Enough theory, let's get started. PetitParser is developed in Pharo, but is also available on other Smalltalk platforms. A ready made image can be downloaded.[1] To load PetitParser into an existing image evaluate the following Gofer expression:

Script 1.1: *Installing PetitParser*

```
Gofer new
  renggli: 'petit';
  package: 'PetitParser';
  package: 'PetitTests';
  load.
```

There are other packages in the same repository that provide additional features, for example PetitSmalltalk is a Smalltalk grammar, PetitXml is an XML grammar, PetitJson is a JSON grammar, PetitAnalyzer provides functionality to analyze and transform grammars, and PetitGui is a Glamour IDE (see Chapter **??**) for writing complex grammars. We are not going to use any of these packages for now.

More information on how to get PetitParser can be found on the website of the project.[2]

[1] http://hudson.lukas-renggli.ch/job/PetitParser/lastSuccessfulBuild/artifact/petitparser
[2] http://scg.unibe.ch/research/helvetia/petitparser

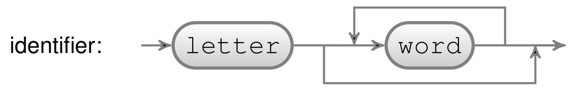

Figure 1.1: Syntax diagram representation for the identifier parser defined in script **??**

Writing a Simple Grammar

Writing grammars with PetitParser is as simple as writing Smalltalk code. For example a grammar parsing identifiers that start with a letter followed by zero or more letters or digits is defined as follows:

Script 1.2: *Creating our first parser to parse identifiers*

```
identifier := #letter asParser , #word asParser star.
```

Figure **??** presents a graphical representation, called a syntax diagram, of the identifier parser. Such a syntax diagram contains squared boxes that represent non terminals (parsers composed of other parsers), round boxes that represent terminals (all other kinds of parsers), arrows between these boxes, some entry points and some exit points. The idea behind this representation is that if an input can go from the entry point to the exit point by following the arrows and being consumed by the boxes, this input is matched by the represented parser.

If you inspect the object identifier you'll notice that it is an instance of a PPSequenceParser. If you dive further into the object you notice the following simple tree of different parser objects:

Script 1.3: *Composition of parsers used for the identifier parser*

```
PPSequenceParser (accepts a sequence of parsers)
   PPPredicateObjectParser (accepts a single letter)
   PPPossessiveRepeatingParser (accepts zero or more instances of another parser)
     PPPredicateObjectParser (accepts a single word character)
```

The root parser is a sequence parser because the #, operator (comma) created a sequence of a letter and zero or more word character parser. The first child of the root parser is a predicate object parser created by the #letter asParser expression. This parser is capable of parsing a single letter as defined by the Character»isLetter method. The second child of the root parser is a repeating parser created by the star call. This parser uses its child parser (another predicate object parser) as much as possible on the input (*i.e.*, it is a *greedy* parser). Its child parser is a predicate object parser created by the #word asParser expression. This parser is capable of parsing a single digit or letter as defined by the Character»isDigit and Character»isLetter methods.

Parsing Some Input

To actually parse a string (or stream) we can use the method PPParser»parse::

Script 1.4: *Parsing some input strings with the identifier parser*

```
identifier parse: 'yeah'.        ⟶   #($y #($e $a $h))
identifier parse: 'f123'.        ⟶   #($f #($1 $2 $3))
```

While it seems odd to get these nested arrays with characters as a return value, this is the default decomposition of the input into a parse tree. We'll see in a while how that can be customized.

If we try to parse something invalid we get an instance of PPFailure as an answer:

Script 1.5: *Parsing invalid input results in a failure*

```
identifier parse: '123'.         ⟶   letter expected at 0
```

This parsing results in a failure because the first character (1) is not a letter. Instances of PPFailure are the only objects in the system that answer with true when you send the message #isPetitFailure. Alternatively you can also use PPParser»parse:onError: to throw an exception in case of an error:

```
identifier
  parse: '123'
  onError: [ :msg :pos | self error: msg ].
```

If you are only interested if a given string (or stream) matches or not you can use the following constructs:

Script 1.6: *Checking that some inputs are identifiers*

```
identifier matches: 'foo'.       ⟶   true
identifier matches: '123'.       ⟶   false
identifier matches: 'foo()'.     ⟶   true
```

The last result can be quite surprising:indeed, a parenthesis is neither a digit nor a letter as was specified by the #word asParser expression. In fact, the identifier parser matches "aa" and this is enough for the matches: call to return true. The result would be similar with the use of parse: which would return #($f #($o $o)). If you want to be sure that the complete input is matched, use PPParser»end:

Script 1.7: *Ensuring that the whole input is matched*

```
identifier end matches: 'foo()'.     ⟶   false
```

The end call creates a new parser that matches the end of input. To be able to compose parsers easily, it is important that parsers do not match the

Terminal Parsers	Description
$a asParser	Parses the character $a.
'abc' asParser	Parses the string 'abc'.
#any asParser	Parses any character.
#digit asParser	Parses one digit (0..9).
#letter asParser	Parses one letter (a..z and A..Z).
#word asParser	Parses a digit or letter.
#blank asParser	Parses a space or a tabulation.
#newline asParser	Parses the carriage return or line feed characters.

Table 1.1: PetitParser pre-defines a multitude of terminal parsers

end of input by default. Because of this, you might be interested to find all the places that a parser can match:

Script 1.8: *Finding all matches in an input*

```
identifier matchesSkipIn: 'foo 123 bar12'.
identifier matchesIn: 'foo 123 bar12'.
identifier matchingSkipRangesIn: 'foo 123 bar12'.
identifier matchingRangesIn: 'foo 123 bar12'.
```

The PPParser»matchesSkipIn: method returns a collection of arrays containing what has been matched (*e.g.*, #($f #($o $o)) and #($b #($a $r $1 $2)) in this case. This function avoids parsing the same character twice. The method PPParser»matchesIn: does a similar job but returns a collection which also contains sub-parsed elements: *e.g.*, evaluating identifier matchesIn: 'foo 123 bar12' returns a collection of 6 elements: #($f #($o $o)),#($o #($o)), #($o #()), #($b #($a $r $1 $2)),#($a #($r $1 $2)), and #($r #($1 $2)). Similarly, to find all the matching ranges (index of first character and index of last character) in the given input one can use either PPParser»matchingSkipRangesIn: or PPParser» matchingRangesIn:: *e.g.*, evaluating identifier matchingSkipRangesIn: 'foo 123 bar12' returns a collection with (1 to: 3) and (9 to: 13).

Different Kinds of Parsers

PetitParser provide a large set of ready-made parser that you can compose to consume and transform arbitrarily complex languages. The terminal parsers are the most simple ones. We've already seen a few of those, some more are defined in Table **??**.

The class side of PPPredicateObjectParser provides a lot of other factory methods that can be used to build more complex terminal parsers. To use them, send the message asParser to a symbol containing the name of the factory method (such as #punctuation asParser).

Parser Combinators	Description
p1 , p2	Parses p1 followed by p2 (sequence).
p1 / p2	Parses p1, if that doesn't work parses p2.
p star	Parses zero or more p.
p plus	Parses one or more p.
p optional	Parses p if possible.
p and	Parses p but does not consume its input.
p negate	Parses p and succeeds when p fails.
p not	Parses p and succeeds when p fails, but does not consume its input.
p end	Parses p and succeeds only at the end of the input.

Table 1.2: PetitParser pre-defines a multitude of parser combinators

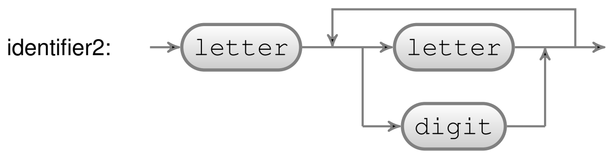

Figure 1.2: Syntax diagram representation for the identifier2 parser defined in script **??**

Action Parsers	Description
p ==> aBlock	Performs the transformation given in aBlock.
p flatten	Creates a string from the result of p.
p token	Similar to flatten but returns a PPToken with more information.
p trim	Trims whitespaces before and after p.

Table 1.3: PetitParser pre-defines a multitude of action parsers

The next set of parsers are used to combine other parsers together and is defined in Table **??**.

So instead of using the #word predicate we could have written our identifier parser like this (see also Figure **??**):

Script 1.9: *Another way of defining the identifier parser*

```
identifier2 := #letter asParser , (#letter asParser / #digit asParser) star.
```

To attach an action or transformation to a parser we can use one of the methods defined in Table **??**.

To return a string of the parsed identifier, we can modify our parser like

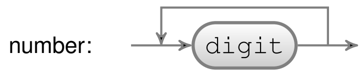

Figure 1.3: Syntax diagram representation for the number parser defined in script **??**

this:

Script 1.10: *Using flatten so that the parsing result is a string*

```
identifier3 := (#letter asParser , (#letter asParser / #digit asParser) star) flatten.
```

These are the basic elements to build parsers. There are a few more well documented and tested factory methods in the operators protocols of PPParser. If you want to know more about these factory methods, browse these protocols.

Writing a More Complicated Grammar

Now we are able to write a more complicated grammar for evaluating simple arithmetic expressions. Within a workspace we start with the grammar for a number (actually an integer):

Script 1.11: *Parsing integers*

```
number := #digit asParser plus token trim ==> [ :token | token value asNumber ].
number parse: '123'          ⟶    123
```

The token call allows to get a token in the action block instead of an array. The trim call allows the parser to accept spaces before and after the number such as in ' 123 '. The action block asks the parser to convert any parsed string (fetched from the token using PPToken»value) to a number using the String»asNumber method.

The next step is to define the productions for addition and multiplication in order of precedence. Note that we instantiate the productions as PPDelegateParser upfront, because they recursively refer to each other. The method #setParser: then resolves this recursion. The following script defines three parsers for the addition, multiplication and parenthesis (see Figure **??** for the related syntax diagram):

Script 1.12: *Parsing arithmetic expressions*

```
term := PPDelegateParser new.
prod := PPDelegateParser new.
prim := PPDelegateParser new.
```

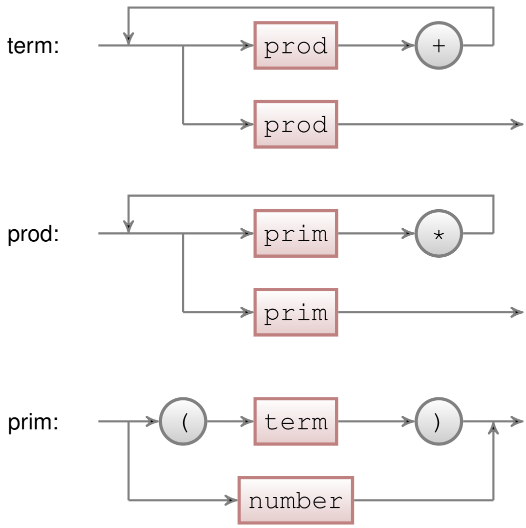

Figure 1.4: Syntax diagram representation for the term, prod, and prim parsers defined in script **??**

```
term setParser: (prod , $+ asParser trim , term ==> [ :nodes | nodes first + nodes last ])
                / prod.
prod setParser: (prim , $* asParser trim , prod ==> [ :nodes | nodes first * nodes last ])
                / prim.
prim setParser: ($( asParser trim , term , $) asParser trim ==> [ :nodes | nodes second ])
                / number.
```

The term parser is defined as being either (1) a prod followed by '+', followed by another term or (2) a prod. In case (1), an action block asks the parser to compute the arithmetic addition of the value of the first node (a prod) and the last node (a term). The prod parser is similar to the term parser. The prim parser is interesting in that it accepts left and right parenthesis before and after a term and has an action block that simply ignores them.

To make sure that our parser consumes all input we wrap it with the end parser into the start production:

```
start := term end.
```

That's it, we can now test our parser:

Script 1.13: *Trying our arithmetic expressions evaluator*

```
start parse: '1 + 2 * 3'.       ⟶  7
start parse: '(1 + 2) * 3'.     ⟶  9
```

1.2 Composite Grammars with PetitParser

In the previous section we saw the basic principles of PetitParser and gave some introductory examples. In this section we are going to present a way to define more complicated grammars. We continue where we left off with the arithmetic expression grammar.

Writing parsers as a script as we did previously can be cumbersome, especially when grammar productions are mutually recursive and refer to each other in complicated ways. Furthermore a grammar specified in a single script makes it unnecessary hard to reuse specific parts of that grammar. Luckily there is PPCompositeParser to the rescue.

1.3 Defining the Grammar

As an example let's create a composite parser using the same expression grammar we built in the last section:

Script 1.14: *Creating a class to hold our arithmetic expression grammar*

```
PPCompositeParser subclass: #ExpressionGrammar
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PetitTutorial'
```

Again we start with the grammar for an integer number. Define the method ExpressionGrammar»number as follows:

Script 1.15: *Implementing our first parser as a method*

```
ExpressionGrammar>>number
  ↑ #digit asParser plus token trim ==> [ :token | token value asNumber ]
```

Every production in ExpressionGrammar is specified as a method that returns its parser. Next we define the productions term, prod, and prim. Productions refer to each other by reading the respective instance variable of the same name. This is important to be able to create recursive grammars. The instance variables themselves are typically not written to as PetitParser takes care to initialize them for you automatically. We let Pharo automatically add the necessary instance variables as we refer to them for the first time.

Script 1.16: *Defining more expression grammar parsers, this time with no associated action*

```
ExpressionGrammar>>term
  ↑ add / prod
```

```
ExpressionGrammar>>add
  ↑ prod , $+ asParser trim , term

ExpressionGrammar>>prod
  ↑ mul / prim

ExpressionGrammar>>mul
  ↑ prim , $* asParser trim , prod

ExpressionGrammar>>prim
  ↑ parens / number

ExpressionGrammar>>parens
  ↑ $( asParser trim , term , $) asParser trim
```

Contrary to our previous implementation we do not define the production actions yet (what we previously did by using PPParser»==>); and we factor out the parts for addition (add), multiplication (mul), and parenthesis (parens) into separate productions. This will give us better reusability later on. Usually, production methods are categorized in a protocol named grammar (which can be refined into more specific protocol names when necessary such as grammar−literals).

Last but not least we define the starting point of the expression grammar. This is done by overriding PPCompositeParser»start in the ExpressionGrammar class:

Script 1.17: *Defining the starting point of our expression grammar parser*

```
ExpressionGrammar>>start
  ↑ term end
```

Instantiating the ExpressionGrammar gives us an expression parser that returns a default abstract-syntax tree:

Script 1.18: *Testing our parser on simple arithmetic expressions*

```
parser := ExpressionGrammar new.
parser parse: '1 + 2 * 3'.        ⟶   #(1 $+ #(2 $* 3))
parser parse: '(1 + 2) * 3'.      ⟶   #(#($( #(1 $+ 2) $)) $* 3)
```

1.4 Defining the Evaluator

Now that we have defined a grammar we can reuse this definition to implement an evaluator. To do this we create a subclass of ExpressionGrammar called ExpressionEvaluator.

Script 1.19: *Separating the grammar from the evaluator by creating a subclass*

```
ExpressionGrammar subclass: #ExpressionEvaluator
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PetitTutorial'
```

We then redefine the implementation of add, mul and parens with our evaluation semantics. This is accomplished by calling the super implementation and adapting the returned parser as follows:

Script 1.20: *Refining the definition of some parsers to evaluate arithmetic expressions*

```
ExpressionEvaluator>>add
  ↑ super add ==> [ :nodes | nodes first + nodes last ]

ExpressionEvaluator>>mul
  ↑ super mul ==> [ :nodes | nodes first * nodes last ]

ExpressionEvaluator>>parens
  ↑ super parens ==> [ :nodes | nodes second ]
```

The evaluator is now ready to be tested:

Script 1.21: *Testing our evaluator on simple arithmetic expressions*

```
parser := ExpressionEvaluator new.
parser parse: '1 + 2 * 3'.      →  7
parser parse: '(1 + 2) * 3'.    →  9
```

Similarly a pretty printer can be defined by subclassing ExpressionGrammar as follows:

Script 1.22: *Separating the grammar from the pretty printer by creating a subclass*

```
ExpressionGrammar subclass: #ExpressionPrinter
 instanceVariableNames: ''
 classVariableNames: ''
 poolDictionaries: ''
 category: 'PetitTutorial'

ExpressionPrinter>>add
 ↑ super add ==> [:nodes | nodes first, ' + ', nodes third]

ExpressionPrinter>>mul
 ↑ super mul ==> [:nodes | nodes first, ' * ', nodes third]

ExpressionPrinter>>number
 ↑ super number ==> [:num | num printString]
```

```
ExpressionPrinter>>parens
  ↑ super parens ==> [:node | '(', node second, ')']
```

This pretty printer can be tried out as follows:

Script 1.23: *Testing our pretty printer on simple arithmetic expressions*

```
parser := ExpressionPrinter new.
parser parse: '1+2 *3'.        −→   '1 + 2 * 3'
parser parse: '(1+ 2 )* 3'.    −→   '(1 + 2) * 3'
```

1.5 Testing a Grammar

The PetitParser contains a framework dedicated to testing your grammars. Testing a grammar is done by subclassing PPCompositeParserTest as follows:

Script 1.24: *Creating a class to hold the tests for our arithmetic expression grammar*

```
PPCompositeParserTest subclass: #ExpressionGrammarTest
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PetitTutorial'
```

It is then important that the test case class references the parser class: this is done by overriding the PPCompositeParserTest»parserClass method in ExpressionGrammarTest:

Script 1.25: *Linking our test case class to our parser*

```
ExpressionGrammarTest>>parserClass
  ↑ ExpressionGrammar
```

Writing a test scenario can be done by implementing new methods in ExpressionGrammarTest:

Script 1.26: *Implementing tests for our arithmetic expression grammar*

```
ExpressionGrammarTest>>testNumber
  self parse: '123 ' rule: #number.

ExpressionGrammarTest>>testAdd
  self parse: '123+77' rule: #add.
```

These tests ensure that the ExpressionGrammar parser can parse some expressions using a specified production rule. Testing the evaluator and pretty printer is similarly easy:

Script 1.27: *Testing the evaluator and pretty printer*

```
ExpressionGrammarTest subclass: #ExpressionEvaluatorTest
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PetitTutorial'

ExpressionEvaluatorTest>>parserClass
  ↑ ExpressionEvaluator

ExpressionEvaluatorTest>>testAdd
  super testAdd.
  self assert: result equals: 200

ExpressionEvaluatorTest>>testNumber
  super testNumber.
  self assert: result equals: 123

ExpressionGrammarTest subclass: #ExpressionPrinterTest
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PetitTutorial'

ExpressionPrinterTest>>parserClass
  ↑ ExpressionPrinter

ExpressionPrinterTest>>testAdd
  super testAdd.
  self assert: result equals: '123 + 77'

ExpressionPrinterTest>>testNumber
  super testNumber.
  self assert: result equals: '123'
```

1.6 Debugging a Grammar

PetitParser comes with tools to help debug a grammar. These tools are mostly written by Tudor Girba using the Glamour framework (see Chapter **??**).

1.7 Example Grammars

In this section we illustrate PetitParser through the presentation of various parsers.

Parsing Java annotations

Parsing Smalltalk

1.8 Chapter Conclusion

This concludes our tutorial of PetitParser. Please note that this tutorial is not meant to give an exhaustive overview of PetitParser, but is merely intended to introduce the reader to the usage and to our intent for our approach. For a more extensive view of PetitParser, its concepts and implementation, the Moose book[3] and Lukas Renggli's PhD have a dedicated chapter dedicated.

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