Chapter 1

PetitParser

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

Building parsers to analyze and transform data is a common task in software development. In this chapter we present a powerful parser framework called PetitParser. PetitParser combines many ideas from various parsing technologies to model grammars and parsers as objects that can be reconfigured dynamically. PetitParser was written by Lukas Renggli as part of his work on the Helvetia system but it can be used as a standalone library.

1.1 Writing Parsers with PetitParser

PetitParser is a parsing framework different from many other popular parser generators. PetitParser makes it easy to define parsers with Smalltalk code and to dynamically reuse, compose, transform and extend grammars. We can reflect on the resulting grammars and modify them on-the-fly. As such PetitParser fits better the dynamic nature of Smalltalk.

Furthermore, PetitParser is not based on tables such as SmaCC and ANTLR. Instead it uses a 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. 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 the notion of ordered choices. Unlike 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 talking, let's get started. PetitParser is developed in Pharo, and there are also versions for Java and Dart available. 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: PetitSmalltalk is a complete 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 1.6) 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].

Writing a Simple Grammar

Writing grammars with PetitParser is as simple as writing Smalltalk code. For example, to define a grammar that parses identifiers starting with a letter followed by zero or more letters or digits is defined and used as follows:

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

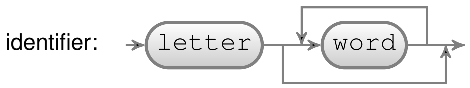

Figure 1.1: Syntax diagram representation for the identifier parser defined in script 1.2

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

```
|identifier|
identifier := #letter asParser , #word asParser star.
identifier parse: 'a987jlkj'    ⟶    #($a #($9 $8 $7 $j $l $k $j))
```

A Graphical Notation

Figure 1.1 presents a syntax diagram of the identifier parser. Each box represents a parser. The arrows between the boxes represent the flow in which input is consumed. The rounded boxes are elementary parsers (terminals). The squared boxes (not shown on this figure) are parsers composed of other parsers (non terminals).

If you inspect the object identifier of the previous script, you'll notice that it is an instance of a PPSequenceParser. If you dive further into the object you notice the following 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 , (comma) operator creates a sequence from (1) a letter parser and (2) zero or more word character parser. The root parser first child 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 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 use the method PPParser»parse: as follows:

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 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 "foo" and this is enough for the PPParser»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 the message PPParser»end as follows:

Script 1.7: *Ensuring that the whole input is matched using* PPParser»end

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

The PPParser»end message 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 end of input by default. Because of this, you might be interested to find all the places that a parser can match using the message PPParser» matchesSkipIn: and PPParser»matchesIn:.

Script 1.8: *Finding all matches in an input*

```
identifier matchesSkipIn: 'foo 123 bar12'.
    ⟶    an OrderedCollection(#($f #($o $o)) #($b #($a $r $1 $2)))

identifier matchesIn: 'foo 123 bar12'.
    ⟶    an OrderedCollection(#($f #($o $o)) #($o #($o)) #($o #()) #($b #($a $r $1 $2))
       #($a #($r $1 $2)) #($r #($1 $2)))
```

The PPParser»matchesSkipIn: method returns a collection of arrays containing what has been matched. This function avoids parsing the same character twice. The method PPParser»matchesIn: does a similar job but returns a collection with all possible sub-parsed elements: *e.g.*, evaluating identifier matchesIn: 'foo 123 bar12' returns a collection of 6 elements.

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: as shown by the script below:

Script 1.9: *Finding all matched ranges in an input*

```
identifier matchingSkipRangesIn: 'foo 123 bar12'.
    ⟶    an OrderedCollection((1 to: 3) (9 to: 13))

identifier matchingRangesIn: 'foo 123 bar12'.
    ⟶    an OrderedCollection((1 to: 3) (2 to: 3) (3 to: 3) (9 to: 13) (10 to: 13) (11 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 the protocol Table 1.1.

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 PPParser»asParser to a symbol containing the name of the factory method (such as #punctuation asParser).

The next set of parsers are used to combine other parsers together and is defined in the protocol Table 1.2.

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.
#space asParser	Parses a space character.
#lowercase asParser	Parses a lowercase character.
#uppercase asParser	Parses an uppercase character.
nil asParser	Parses nothing.

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

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.
p times: n	Parses p exactly n time.
p min: n max: m	Parses p at least n times up to m times
p starLazy: q	Like star but stop consumming when q succeed

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

As a simple example of parser combination, the following definition of the identifier2 parser is equivalent to our previous definition of identifier:

Script 1.10: *A different way to express the* identifier *parser*

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

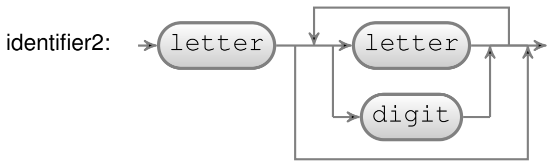

Figure 1.2: Syntax diagram representation for the identifier2 parser defined in script 1.10

Parser Action

To define an action or transformation on a parser we can use one of the messages PPParser»==>, PPParser»flatten, PPParser»token and PPParser»trim defined in the protocol Table 1.3.

Action Parsers	Description
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 white spaces before and after p.
p ==> aBlock	Performs the transformation given in aBlock.

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

To return a string of the parsed identifier instead of getting an array of matched elements, configure the parser by sending it the message PPParser» flatten.

Script 1.11: *Using flatten so that the parsing result is a string*

```
|identifier|
identifier := (#letter asParser , (#letter asParser / #digit asParser) star).
identifier parse: 'ajka0'          ⟶     #($a #($j $k $a $0))

identifier flatten parse: 'ajka0'          ⟶     'ajka0'
```

The message PPParser»token is similar to flatten but returns a PPToken that provide much more contextual information like the collection where the token was located and its position in the collection.

Sending the message PPParser»trim configures the parser to ignore white spaces at the beginning and end of the parsed result. In the following, using the first parser on the input leads to an error because the parser does not accept the starting spaces. With the second parser, spaces are ignored.

Script 1.12: *Using trim to remove spaces*

```
|identifier|
```

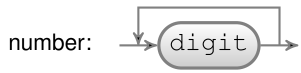

Figure 1.3: Syntax diagram representation for the number parser defined in script 1.13

```
identifier := (#letter asParser , #word asParser star) flatten.
identifier parse: '  ajka   '             ⟶    letter expected at 0

identifier trim parse: '  ajka   '         ⟶    'ajka'
```

The message PPParser»==> lets you specify a block to be executed when the parser matches an input. The next section presents several examples. Here is a simple way to get a number from its string representation.

Script 1.13: *Parsing integers*

```
number := #digit asParser plus flatten ==> [ :str | str asNumber ].
number parse: '123'                        ⟶    123
```

The table 1.3 shows 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. An interesting one is separatedBy: which answers a new parser that parses the input one or more times, with separations specified by another parser.

Writing a More Complicated Grammar

We now write a more complicated grammar for evaluating simple arithmetic expressions. With the grammar for a number (actually an integer) defined above, 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 1.4 for the related syntax diagram):

Script 1.14: *Parsing arithmetic expressions*

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

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

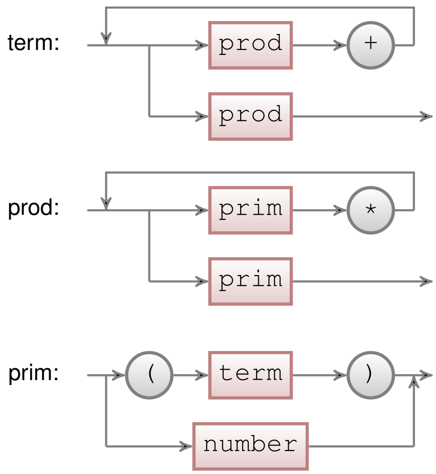

Figure 1.4: Syntax diagram representation for the term, prod, and prim parsers defined in script 1.14

```
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 understand the precedence of productions, see Figure 1.5. The root of the tree in this figure (term), is the production that is tried first. A term is either a + or a prod. The term production comes first because + as the lowest priority in mathematics.

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 1.15: *Trying our arithmetic expressions evaluator*

```
start parse: '1 + 2 * 3'.     →   7
```

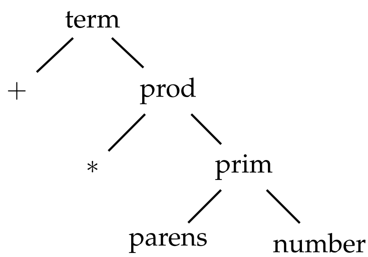

Figure 1.5: Explains how to understand the precedence of productions. An expression is a term which is either a sum or a production. It is necessary to recognize sums first as they have the lowest priority. A production is either a multiplication or a primitive. A primitive is either a parenthesised expression or a number.

```
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.

Defining the Grammar

As an example let's create a composite parser using the same expression grammar we built in the last section but this time we define it inside a class subclass of PPCompositeParser.

Script 1.16: *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 number as follows:

Script 1.17: *Implementing our first parser as a method*

```
ExpressionGrammar>>number
  ^ #digit asParser plus flatten trim ==> [ :str | str asNumber ]
```

Every production in ExpressionGrammar is specified as a method that returns its parser. Similarly, we define the productions term, prod, mul, and prim. Productions refer to each other by reading the respective instance variable of the same name and PetitParser takes care of initializing these instance variables for you automatically. We let Pharo automatically add the necessary instance variables as we refer to them for the first time. We obtain the following class definition:

Script 1.18: *Creating a class to hold our arithmetic expression grammar*

```
PPCompositeParser subclass: #ExpressionGrammar
  instanceVariableNames: 'add prod tern mul parens number'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PetitTutorial'
```

Script 1.19: *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. For example, a subclass may override such methods to produce slightly different production output. 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 1.20: *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 1.21: *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)
```

Defining an 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 1.22: *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 shown in the following methods.

Script 1.23: *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 1.24: *Testing our evaluator on simple arithmetic expressions*

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

Defining a Pretty-Printer

We can reuse the grammar for example to define a simple pretty printer. This is as easy as subclassing ExpressionGrammar again!

Script 1.25: *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 shown by the following expressions.

Script 1.26: *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'
```

Easy expressions with PPExpressionParser

PetitParser propose a powerful tool to create expressions; PPExpressionParser is a parser to conveniently define an expression grammar with prefix, postfix, and left- and right-associative infix operators. The operator-groups are defined in descending precedence.

Script 1.27: *The ExpressionGrammar we define previously can be implemented in few lines*

```
|expression parens number|
expression := PPExpressionParser new.
parens := $( asParser token trim , expression , $) asParser token trim
==> [ :nodes | nodes second ].
number := #digit asParser plus flatten trim ==> [ :str | str asNumber ].

expression term: parens / number.

expression
group: [ :g |
g left: $* asParser token trim do: [ :a :op :b | a * b ].
g left: $/ asParser token trim do: [ :a :op :b | a / b ] ];
group: [ :g |
g left: $+ asParser token trim do: [ :a :op :b | a + b ].
g left: $− asParser token trim do: [ :a :op :b | a − b ] ].
```

Script 1.28: *Now our parser is also able to manage subtraction and division*

```
expression parse: '1−2/3'.    ⟶    (1/3)
```

1.3 Testing a Grammar

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

Script 1.29: *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 1.30: *Linking our test case class to our parser*

```
ExpressionGrammarTest>>parserClass
  ^ ExpressionGrammar
```

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

Script 1.31: *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 1.32: *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'
```

1.4 Case Study: A JSON Parser

In this section we illustrate PetitParser through the development of a JSON parser. JSON is a lightweight data-interchange format defined in http://www.json.org. We are going to use the specification on this website to define our own JSON parser.

JSON is a simple format based on nested pairs and arrays. The following script gives an example taken from Wikipedia http://en.wikipedia.org/wiki/JSON

Script 1.33: *An example of JSON*

```
{   "firstName" : "John",
    "lastName" : "Smith",
    "age" : 25,
    "address" :
       {   "streetAddress" : "21 2nd Street",
           "city" : "New York",
           "state" : "NY",
           "postalCode" : "10021"  },
    "phoneNumber":
       [
          {   "type" : "home",
              "number" : "212 555−1234"   },
          {   "type" : "fax",
              "number" : "646 555−4567"   }   ]   }
```

JSON consists of object definitions (between curly braces "{}") and arrays (between square brackets "[]"). An object definition is a set of key/value pairs whereas an array is a list of values. The previous JSON example then represents an object (a person) with several key/value pairs (*e.g.*, for the person's first name, last name, and age). The address of the person is represented by another object while the phone number is represented by an array of objects.

First we define a grammar as subclass of PPCompositeParser. Let us call it PPJsonGrammar

Script 1.34: *Defining the JSON grammar class*

```
PPCompositeParser subclass: #PPJsonGrammar
  instanceVariableNames: ''
```

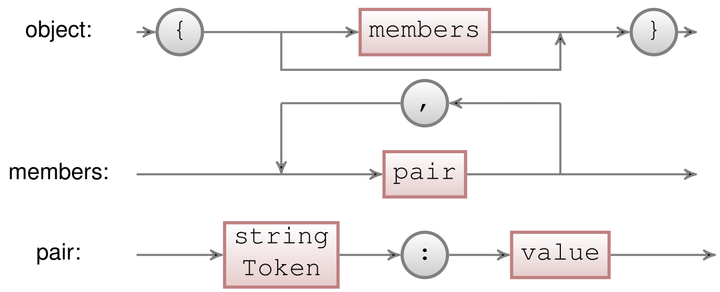

Figure 1.6: Syntax diagram representation for the JSON object parser defined in script 1.35

```
classVariableNames: 'CharacterTable'
poolDictionaries: ''
category: 'PetitJson-Core'
```

We define the CharacterTable class variable since we will later use it to parse strings.

Parsing Objects and Arrays

The syntax diagrams for JSON objects and arrays are in Figure 1.6 and Figure 1.7. A PetitParser can be defined for JSON objects with the following code:

Script 1.35: *Defining the JSON parser for object as represented in Figure 1.6*

```
PPJsonGrammar>>object
  ^ ${ asParser token trim , members optional , $} asParser token trim

PPJsonGrammar>>members
  ^ pair separatedBy: $, asParser token trim

PPJsonGrammar>>pair
  ^ stringToken , $: asParser token trim , value
```

The only new thing here is the call to the PPParser»separatedBy: convenience method which answers a new parser that parses the receiver (a value here) one or more times, separated by its parameter parser (a comma here).

Arrays are much simpler to parse as depicted in the script 1.36.

Script 1.36: *Defining the JSON parser for array as represented in Figure 1.7*

```
PPJsonGrammar>>array
  ^ $[ asParser token trim ,
```

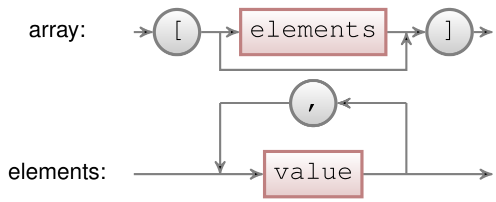

Figure 1.7: Syntax diagram representation for the JSON array parser defined in script 1.36

```
    elements optional ,
  $] asParser token trim

PPJsonGrammar>>elements
 ^ value separatedBy: $, asParser token trim
```

Parsing Values

In JSON, a value is either a string, a number, an object, an array, a Boolean (true or false), or null. The value parser is defined as below and represented in Figure 1.8:

Script 1.37: *Defining the JSON parser for value as represented in Figure 1.8*

```
PPJsonGrammar>>value
 ^ stringToken / numberToken / object / array /
   trueToken / falseToken / nullToken
```

A string requires quite some work to parse. A string starts and end with double-quotes. What is inside these double-quotes is a sequence of characters. Any character can either be an escape character, an octal character, or a normal character. An escape character is composed of a backslash immediately followed by a special character (*e.g.*, '\n' to get a new line in the string). An octal character is composed of a backslash, immediately followed by the letter 'u', immediately followed by 4 hexadecimal digits. Finally, a normal character is any character except a double quote (used to end the string) and a backslash (used to introduce an escape character).

Script 1.38: *Defining the JSON parser for string as represented in Figure 1.9*

```
PPJsonGrammar>>stringToken
 ^ string token trim
PPJsonGrammar>>string
 ^ $" asParser , char star , $" asParser
PPJsonGrammar>>char
```

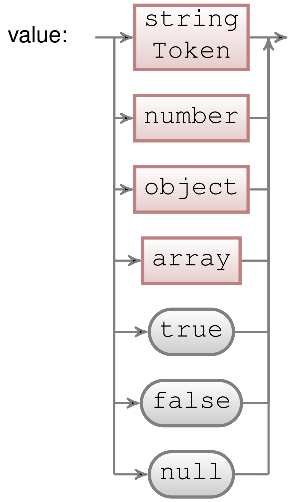

Figure 1.8: Syntax diagram representation for the JSON value parser defined in script 1.37

```
  ^ charEscape / charOctal / charNormal
PPJsonGrammar>>charEscape
  ^ $\ asParser , (PPPredicateObjectParser anyOf: (String withAll: CharacterTable keys))
PPJsonGrammar>>charOctal
  ^ '\u' asParser , (#hex asParser min: 4 max: 4)
PPJsonGrammar>>charNormal
  ^ PPPredicateObjectParser anyExceptAnyOf: '\"'
```

Special characters allowed after a slash and their meanings are defined in the CharacterTable dictionary that we initialize in the initialize class method.

Script 1.39: *Defining the JSON special characters and their meaning*

```
PPJsonGrammar class>>initialize
  CharacterTable := Dictionary new.
  CharacterTable
    at: $\ put: $\;
    at: $/ put: $/;
    at: $" put: $";
    at: $b put: Character backspace;
    at: $f put: Character newPage;
    at: $n put: Character lf;
    at: $r put: Character cr;
    at: $t put: Character tab
```

Parsing numbers is only slightly simpler as a number can be positive or negative and integral or decimal. Additionally, a decimal number can be

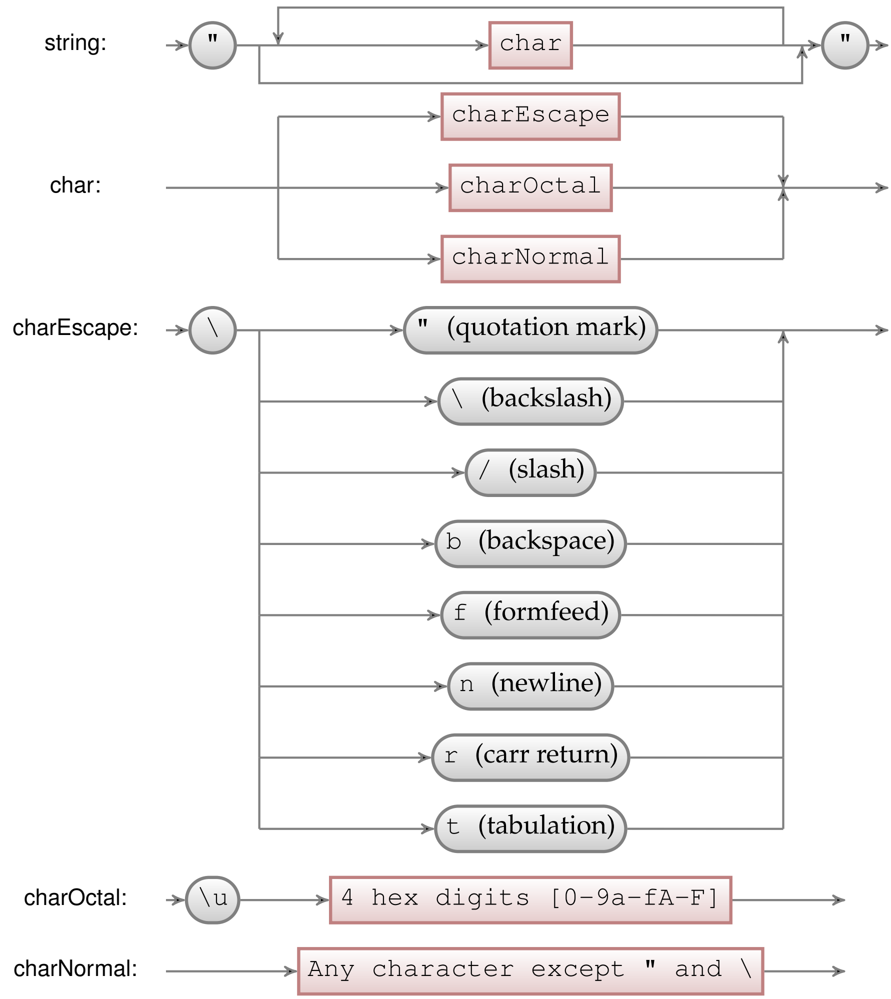

Figure 1.9: Syntax diagram representation for the JSON string parser defined in script 1.38

expressed with a floating number syntax.

Script 1.40: *Defining the JSON parser for number as represented in Figure 1.10*

```
PPJsonGrammar>>numberToken
  ^ number token trim
PPJsonGrammar>>number
  ^ $− asParser optional ,
  ($0 asParser / #digit asParser plus) ,
  ($. asParser , #digit asParser plus) optional ,
```

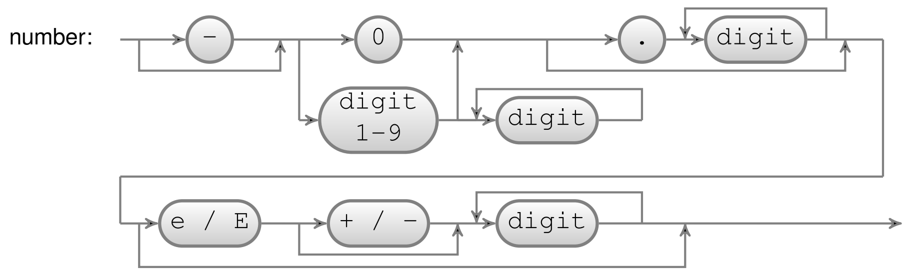

Figure 1.10: Syntax diagram representation for the JSON number parser defined in script 1.40

```
(($e asParser / $E asParser) , ($− asParser / $+ asParser) optional , #digit asParser
    plus) optional
```

The attentive reader will have noticed a small difference between the syntax diagram in Figure 1.10 and the code in script 1.40. Numbers in JSON can not contain leading zeros: *i.e.*, strings such as *"01"* do not represent valid numbers. The syntax diagram makes that particularly explicit by allowing either a 0 or a digit between 1 and 9. In the above code, the rule is made implicit by relying on the fact that the parser combinator $/ is ordered: the parser on the right of $/ is only tried if the parser on the left fails: thus, ($0 asParser / #digit asParser plus) defines numbers as being just a 0 or a sequence of digits not starting with 0.

The other parsers are fairly trivial:

Script 1.41: *Defining missing JSON parsers*

```
PPJsonGrammar>>falseToken
  ^ 'false' asParser token trim
PPJsonGrammar>>nullToken
  ^ 'null' asParser token trim
PPJsonGrammar>>trueToken
  ^ 'true' asParser token trim
```

The only piece missing is the start parser.

Script 1.42: *Defining the JSON start parser as being a value (Figure 1.8) with nothing following*

```
PPJsonGrammar>>start
  ^ value end
```

1.5 PetitParser Browser

PetitParser is shipped with powerful browser that can help you with development of complex parsers. Browser provides graphical visualization, debugging support, refactoring support and some other features discussed later in this chapter. You will see that these features could be very useful while developing your own parser.

Loading PetitParser Browser

Before we can proceed with using the GUI tool we need to load PetitParser with the Glamour IDE. Simply evaluate following expression:

Script 1.43: *Installing PetitParser with GUI tool*

```
Gofer new
  renggli: 'petit';
  package: 'ConfigurationOfPetitParser';
  load.

ConfigurationOfPetitParser loadDefault.
```

Now, we can open the PetitParser browser with this expression:

Script 1.44: *Opening PetitParser browser*

```
PPBrowser open.
```

PetitParser Browser Overview

In Figure 1.11 you can see window of PPBrowser. The left panel called Parsers contains list of all parsers in the system. You can see our ExpressionGrammar and its subclasses as well as PPJsonGrammar we defined earlier in this chapter. By selecting one of the parser you activate upper-right side of the browser. For each rule (*e.g.*, prim) you can see 5 tabs related to the rule.

Source shows source code of the rule. The code can be updated and saved in this window. Moreover, you can add a new rule simply by defining new method name and its body.

Graph shows graphical representation of the rule. It is updated as the rule source is changed.

Example shows automatically generated example based on the definition of the rule. This is useful for verification of your rule definition. Note the

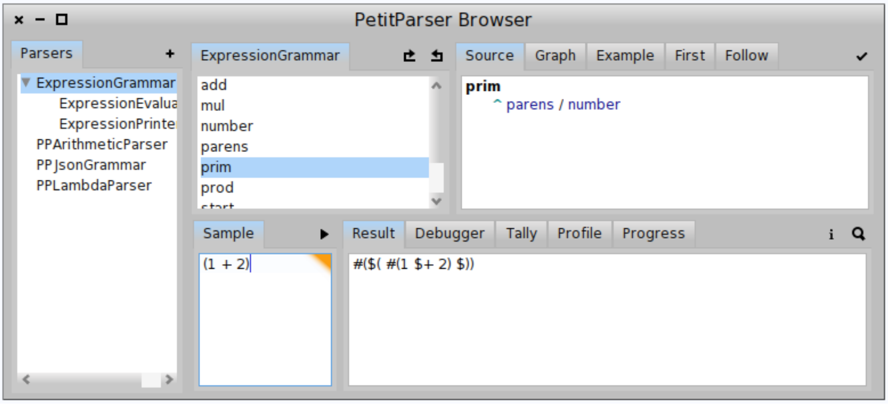

Figure 1.11: PetitParser Browser window.

Reload button (TODO how to refer to button? Use "", ″, italics?, ...) in top right corner that regenerates the example.

First shows list of parsers that can be activated directly after the rule started.

Follow shows list of parsers that can be activated directly after the rule finished.

The lower-right side of the browser is related to particular parsing input. You can specify the input by filling in the text area in the Sample tab. One may execute parsing by clicking play button or by pressing cmd-s. After parsing you may inspect following:

Result shows result of the parsing. Result can be inspected or explored by clicking *Inspect* or *Explore* button (TODO how to refer to button? Use "", ″, italics?, ...).

Debugger shows tree view of steps that were performed during parsing. This is very useful if you don't know what exactly is happening during parsing. By selecting the step the subset of input is highlighted, so you can see which part of input was parsed by particular step. For example, you can inspect how does the PPJsonGrammar work, what rules and in which order are called as is depicted in Figure 1.12.

Tally shows how many times was parser called during parsing of the input. The percentage shows number of calls to total number of calls ratio. This might be useful while optimizing performance of your parser.

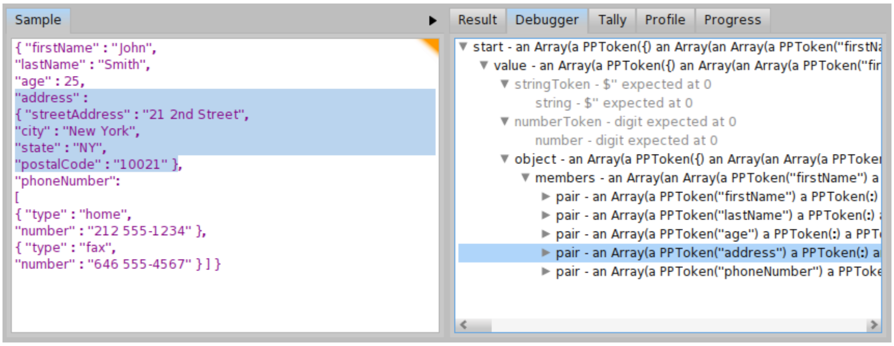

Figure 1.12: Debugger output of PPJsonGrammar.

Figure 1.13: Progress of Petit Parser with a lot of backtracking.

Profile shows how much time was spend in a particular parser during parsing of the input. The percentage shows time to total time ratio. This might be useful while optimizing performance of your parser.

Progress shows position in a string on the x-axis and time on the y-axis. The best case is when the x position is never or only a little decreased. If you see too many backwards jumps *e.g.*, as depicted in Figure 1.13, your parser performs a lot of backtracking which degrades the performance. In that case, you should reconsider the order of choice parsers or restructuralize your grammar.

1.6 Chapter Conclusion

This concludes our introductionary tutorial of PetitParser. For a more extensive view of PetitParser, its concepts and implementation, the Moose book[3] and Lukas Renggli's PhD[4] both have a dedicated chapter.

[4] http://www.themoosebook.org/book/internals/petit-parser
[4] http://scg.unibe.ch/archive/phd/renggli-phd.pdf