PetitParser tutorial

Introduction

with the participation of:

Jan Kurs (kurs@iam.unibe.ch)

Guillaume Larcheveque (guillaume.larcheveque@gmail.com)

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 1 but it can be used as a standalone library.

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.

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:

```
identifier := #letter asParser , #word asParser star.
identifier parse: 'a987jlkj'
```

Parsing some input

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

```
identifier parse: 'yeah'
```

```
identifier parse: 'f123'
```

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:

```
identifier parse: '123'
```

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:

```
identifier matches: 'foo'
```

```
identifier matches: '123'
```

```
identifier matches: 'foo()'
```

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:

```
identifier end matches:'foo'
```

The PPParser»end message creates a new parser that matches the end ofinput. 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:.

```
identifier matchesSkipIn: 'foo 123 bar12'
```

```
identifier matchesIn: 'foo 123 bar12'
```

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 andindex of last character) in the given input one can use either PPParser»matchingSkipRangesIn: or PPParser»matchingRangesIn: as shown by the script below:

```
identifier matchingSkipRangesIn: 'foo 123 bar12'
```

```
identifier matchingRangesIn: 'foo 123 bar12'
```

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 isdefined in the protocol:

Terminal Parsers

```
$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 any white space character including new line.

`#tab asParser`

. Parses a tab character.

`#lowercase asParser`

. Parses a lowercase character.

`#uppercase asParser`

. Parses an uppercase character.

`nil asParser`

. Parses nothing.

Parser Combinators

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 times.

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 suc-ceeds

As a simple example of parser combination, the following definition of the

identifier2 parser is equivalent to our previous definition of *identifier*:

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

To define an action or transformation on a parser we can use one of the mes-sages PPParser»==>, PPParser»flatten, PPParser»token and PPParser»trim defined in the protocol:

Action parsers

p flatten. Creates a string from the result of p.

p token. Similar to flatten but returns a PPToken with details.

p trim. Trims white spaces before and after p.

p trim: trimParser. Trims whatever trimParser can parse (e.g., comments).

p ==> aBlock. Performs the transformation given in aBlock.

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.

```
|identifier|
identifier := (#letter asParser , (#letter asParser / #digit asParser) star).
identifier parse: ' ajka0 '
```

```
|identifier|
identifier := (#letter asParser , (#letter asParser / #digit asParser) star).
identifier parse: 'ajka0'
```

Sending the message trim is equivalent to calling PPParser»trim: with

```
#space asParser
```

as a parameter. That means *trim:* can be useful to ignore other data from the input, source code comments for example:

```
| identifier comment ignorable line |
identifier := (#letter asParser , #word asParser star) flatten.
comment := '//' asParser, #newline asParser negate star.
ignorable := comment / #space asParser.
line := identifier trim: ignorable.
line parse: '// This is a comment
oneIdentifier // another comment'
```

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.

```
number := #digit asParser plus flatten
```

```
number parse: '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):

```
number := #digit asParser plus flatten ==> [:node | node asNumber].
term := PPDelegateParser new.
prod := PPDelegateParser new.
prim := PPDelegateParser new.
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 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:

```
start parse: '1 + 2 * 3'
```

```
start parse: '(1 + 2) * 3'
```

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 rammar.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.

ExpressionGrammar

Again we start with the grammar for an integer number. Define themethod number as follows:

ExpressionGrammar»number

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 Smalltalk automatically add the necessary instance variables as we refer to them for the first time. We obtain the following class definition:

ExpressionGrammar

Define more expression grammar parsers, this time with no associated action:

ExpressionGrammar»term

ExpressionGrammar»add

ExpressionGrammar»prod

ExpressionGrammar»mul

ExpressionGrammar»prim

ExpressionGrammar»parens

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:

ExpressionGrammar»start

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

```
parser := ExpressionGrammar new.

parser parse: '1 + 2 * 3'

parser parse: '(1 + 2) * 3'
```

Writing dependent grammars

You can easily reuse parsers defined by other grammars. For example, imagine you want to create a new grammar that reuses the definition of number inthe ExpressionGrammar we have just defined. For this, you have to declare adependency to ExpressionGrammar:

```
PPCompositeParser subclass: #MyNewGrammar
instanceVariableNames: 'number'
classVariableNames: ''
poolDictionaries: ''
category: 'PetitTutorial'

MyNewGrammar class>>dependencies
"Answer a collection of PPCompositeParser classes that this parser directly
dependends on."
^ {ExpressionGrammar}

MyNewGrammar>>number
"Answer the same parser as ExpressionGrammar>>number."
^ (self dependencyAt: ExpressionGrammar) number
```

Defining an evaluator

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

ExpressionEvaluator:

ExpressionEvaluator

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

ExpressionEvaluator»add

ExpressionEvaluator»mul

ExpressionEvaluator»parens

The evaluator is now ready to be tested:

```
parser := ExpressionEvaluator new.

parser parse: '1 + 2 * 3'.

parser parse: '(1 + 2) * 3'.
```

Defining a Pretty-Printer

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

ExpressionPrinter

ExpressionPrinter»add

ExpressionPrinter»mul

ExpressionPrinter»parens

This pretty printer can be tried out as shown by the following expressions:

```
parser := ExpressionPrinter new.

parser parse: '1+2 *3'.

parser parse: '(1+ 2   )* 3'.
```

Easy expressions with PPExpressionParser

PetitParser proposes a powerful tool to create expressions; PPExpressionParseris 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.

The ExpressionGrammar we previously defined can be implemented infew 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 ] ].
```

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

```
expression parse: '1 - 2/3'.
```

How do you decide when to create a subclass of PPCompositeParser orinstantiate PPExpressionParser ? On the one hand, you should instantiate aPPExpression-Parser if you want to do a small parser for a small task. On theother hand, if you have a grammar that's composed of many parsers, youshould subclass PPCompositeParser .