Writing Parsers with PetitParser

PetitParser 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:

1. *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.
2. *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.
3. *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.
4. *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 here. To load PetitParser into an existing image evaluate the following Gofer expression:

```
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 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.

Writing a Simple Grammar

Writing grammars with PetitParser is simple as writing Smalltalk code. For example to write a grammar that can parse identifiers that start with a letter followed by zero or more letter or digits is defined as follows. In a workspace we evaluate:

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

If you inspect the object `identifier` you'll notice that it is an instance of a `PPSequenceParser`. This is because the `#,` operator created a sequence of *a letter* and *a zero or more word character* parser. If you dive further into the object you notice the following simple

composition of different parser objects:

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

Parsing Some Input

To actually parse a string (or stream) we can use the method `#parse:`:

```
identifier parse: 'yeah'.          " --> #($y #($e $a $h)) "
identifier parse: 'f12'.           " --> #($f #($1 $2)) "
```

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'.           " --> letter expected at 0 "
```

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 `#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'.         " --> true "
identifier matches: '123'.         " --> false "
```

Furthermore to find all matches in a given input string (or stream) you can use:

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

Similarly, to find all the matching ranges in the given input string (or stream) you can use:

```
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:

Terminal Parsers	Description
`$a asParser`	Parses the character `$a`.
`'abc' asParser`	Parses the string `'abc'`.
`#any asParser`	Parses any character.
`#digit asParser`	Parses the digits 0..9.
`#letter asParser`	Parses the letters a..z and A..Z.

The class side of `PPPredicateObjectParser` provides a lot of other factory methods that can be used to build more complex terminal parsers.

The next set of parsers are used to combine other parsers together:

Parser Combinators	Description
`p1 , p2`	Parses `p1` followed by `p2` (sequence).
`p1 / p2`	Parses `p1`, if that doesn't work parses `p2` (ordered choice).
`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 not`	Parses `p` and succeed when `p` fails, but does not consume its input.
`p end`	Parses `p` and succeed at the end of the input.

So instead of using the `#word` predicated we could have written our identifier parser like this:

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

To attach an action or transformation to a parser we can use the following methods:

Action Parsers	Description
`p ==> aBlock`	Performs the transformation given in `aBlock`.
`p flatten`	Creates a string from the result of `p`.
`p token`	Creates a token from the result of `p`.
`p trim`	Trims whitespaces before and after `p`.

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

```
identifier := (#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 `operations` protocol of `PPParser`. If you want browse that protocol.

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):

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

Then we define the productions for addition and multiplication in order of precedence. Note that we instantiate the productions as `PPUnresolvedParser` upfront, because they recursively refer to each other. The method `#def:` resolves this recursion using the reflective facilities of the host language:

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

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

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, now we can test our parser and evaluator:

```
start parse: '1 + 2 * 3'.       " --> 7 "
start parse: '(1 + 2) * 3'.     " --> 9 "
```

As an exercise we could extend the parser to also accept negative numbers and floating point numbers, not only integers. Furthermore it would be useful to add support subtraction and division as well. All these features can be added with a few lines of PetitParser code.

Composite Grammars with PetitParser

In a previous post I described the basic principles of PetitParser and gave some introductory examples. In this blog post I am going to present a way to define more complicated grammars. We continue where we left off the last time, with the expression grammar.

Writing parsers as a script as we did in the previous post can be cumbersome, especially if grammar productions that 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 blog post:

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

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

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

Every production in `ExpressionGrammar` is specified as a method that returns its parser.

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.

Next we define the productions `term`, `prod`, and `prim`. Contrary to our previous implementation we do not define the production actions yet; 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. We let Pharo automatically add the necessary instance variables as we refer to them for the first time.

```
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
```

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

```
ExpressionGrammar>>start
   ^ term end
```

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

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

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`

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

and we redefine the implementation of `add`, `mul` and `parens` with our evaluation semantics:

```
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:

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

Similarly — as an exercise — a pretty printer can be defined by subclassing `ExpressionGrammar` and by redefining a few of its productions:

```
parser := ExpressionPrinter new.
parser parse: '1+2 *3'.          " --> '1 + 2 * 3' "
parser parse: '(1+ 2 )* 3'.      " --> '(1 + 2) * 3' "
```

YOU CAN FIND THIS MATERIAL ONLINE HERE:

http://www.lukas-renggli.ch/blog/petitparser-1/

http://www.lukas-renggli.ch/blog/petitparser-2/