Parsing

Updated Jun 19, 2011 by michael....@gmail.com

This package implements PEG parsing support for Xtreams. The package includes some sample grammars that can be readily used (e.g. PEG, Smalltalk, JSON, XML, Wiki, etc). The grammars are conventionally stored in class methods on Parser in protocol 'grammars'.

A Parser is created from a PEG grammar by parsing the grammar rules using the ParserParser.

```
wikiGrammar := PEG.Parser grammarWiki reading.
wikiParser := PEG.Parser parserPEG parse: 'Grammar' stream: wikiGrammar actor: ParserParser new.
```

The parser then can be applied in the same fashion using arbitrary Actor to act on the parsing results. In the following example we'll use the wiki parser to generate an XML document from a text marked up with wiki syntax (The WikiGenerator class can be found in the parsing tests package).

```
input := 'Single paragraph with *bold* and _italic_ text and a [link]' reading.
wikiParser parse: 'Page' stream: input actor: PEG.WikiGenerator new
```

We'll describe Actor creation using the simple arithmetic expression grammar example from [http://en.wikipedia.org/wiki/Parsing_expression_grammar wikipedia]

```
grammar := '
        Number                <- [0-9]+
        Parenthesised      <- "(" Expr ")"
        Value                   <- Number / Parenthesised
        Product                <- Value (("*" / "/") Value)*
        Sum                     <- Product (("+" / "-") Product)*
        Expr                    <- Sum
         '.
```

As was shown above, the parser is created by parsing the grammar text and using the ParserParser as the actor.

```
parser := PEG.Parser parserPEG
                        parse: 'Grammar'
                        stream: grammar
                        actor: ParserParser new.
```

The resulting parser object can be immediately used to parse expressions (note that the grammar as written does not support whitespace)

```
parser parse: 'Expr' stream: '3+4*(4-3)' actor: nil
```

This will yield the following (roughly formatted to show some semblance of the parse tree of the expression sample)

```
#(#(OrderedCollection ($3 "16r0033") OrderedCollection ())
OrderedCollection (#
        ('+' #(OrderedCollection ($4 "16r0034")
                OrderedCollection (
                        #('*' #(
                                '('
                                        #(#(OrderedCollection ($4 "16r0034") OrderedCollection ())
                                        OrderedCollection (#
                                                ('-' #(OrderedCollection ($3 "16r0033") OrderedCollection ())),
                                ')' )))))))
```

Let's say we'd like to evaluate the expression and return the result. While it's possible, rather than taking the above output apart and working with the pieces, it's easier to create an Actor that will will receive rule specific callbacks and transform the output on the fly. To do that we'll make a subclass of Actor. Let's call it ArithmeticEvaluator. To make the job easier, the Actor class provides a pragma based mechanism for expressing the rule specific callbacks. All it takes is writing a method for each rule where we want to transform the output. The method is linked to the rule by a pragma and the corresponding action is the method body. Here is an example method for the Number rule that turns the digit collection into a number:

```
ArithmeticEvaluator>>Number: digits
	<action: 'Number'>
	^digits inject: 0 into: [ :total :digit | total * 10 + ('0123456789' indexOf: digit) - 1 ]
```

As soon as we have this method we can parse simple numbers

```
parser parse: 'Number' stream: '3456' reading actor: ArithmeticEvaluator new
```

Now let's try to multiply. We'll need a method for the Product rule, but first we need to discuss the input into the callbacks in a bit more detail. The format of the input depends on the rule definition, specifically on the terms of the right side of the rule. The Product rule is defined as follows

```
Product <- Value (("*" / "/") Value)*
```

This means that you'll get an array of whatever comes out of Value and then zero or more pairs of string * or / and another Value result. The "zero or more" (star) operator always yields a collection (possibly empty). The easiest way to figure out the specifics is to add the rule with just a halt in it and then examining what comes in, for example:

```
parser parse: 'Product' stream: '3*4*5*6*7' reading actor: ArithmeticEvaluator new
```

yields this

```
#(	3
	OrderedCollection (
		#('*' 4)
		#('*' 5)
		#('*' 6)
		#('*' 7)))
```

So Product will get a two element Array where the first element is the first Value and the second is a collection of pairs (possibly empty), where each pair is either * or / and then another Value. That's easy to deal with:

```
Product: terms
	<action: 'Product'>
	^terms last inject: terms first into: [ :total :pair |
		(pair first = '*')
			ifTrue: [ total * pair last ]
			ifFalse: [ total / pair last ] ]
```

The Actor pragmas provide another convenience for taking more complex input apart. You can also write the above as

```
Product: first rest: pairs
	<action: 'Product' arguments: #(1 2)>
	^pairs inject: first into: [ :total :pair |
		(pair first = '*')
			ifTrue: [ total * pair last ]
			ifFalse: [ total / pair last ] ]
```

The **arguments:** keyword simply says invoke this method with elements 1 and 2 of the rule input (terms) as arguments. Obviously the method must take the corresponding number of arguments as well. With the Product action in place the sample input should yield 2520. The Sum rule handling is analogous:

```
Sum: first rest: pairs
	<action: 'Sum' arguments: #(1 2)>
	^pairs inject: first into: [ :total :pair |
		(pair first = '+')
			ifTrue: [ total + pair last ]
			ifFalse: [ total - pair last ] ]
```

The last thing we need to deal with are parentheses. That turns out to be rather simple. The rule looks like this:

```
Parenthesised <- "(" Expr ")"
```

So the callback will get a 3 element array of left bracket, an Expression value and a right bracket. All we really need to do is strip the brackets and return the Expr value.

```
Parenthesised: expression
	<action: 'Parenthesised' arguments: #(2)>
	^expression
```

And that's it. We covered the whole grammar. So let's run our actor through a final test example.

```
parser parse: 'Expr' stream: '((3+4)*2-4)/2' actor: ArithmeticEvaluator new
```

Hopefully the result of the above is 5.

Enter a comment: Hint: You can use Wiki Syntax.

Submit