MOUSE: FROM PARSING EXPRESSIONS TO A PRACTICAL PARSER

Version 1.5.1

Roman R. Redziejowski

Parsing Expression Grammar (PEG) is a new way to specify recursive-descent parsers with limited backtracking. The use of backtracking lifts the $LL(1)$ restriction usually imposed by top-down parsers. In addition, PEG can define parsers with integrated lexing.

Mouse is a tool to transcribe PEG into an executable parser written in Java. Unlike some existing PEG generators (e.g., *Rats!*), *Mouse* does not produce a storage-hungry "packrat parser", but a collection of transparent recursive procedures.

An integral feature of *Mouse* is the mechanism for specifying semantics (also in Java). This makes *Mouse* a convenient tool if one needs an ad-hoc language processor. Being written in Java, the processor is operating-system independent.

This is a user's manual in the form of a tutorial that introduces the reader to *Mouse* through a hands-on experience.

January 6, 2012

Copyright © 2009, 2010, 2011, 2012 by Roman R. Redziejowski (www.romanredz.se).

The author gives unlimited permission to copy, translate and/or distribute this document, with or without modifications, as long as this notice is preserved, and information is provided about any changes.

The document is available from http://mousepeg.sourceforge.net.

Contents

Changes from version of November 5, 2011:

No new or modified functionality, only new version number. (The new version contains a bug fix.)

Changes from version of October 15, 2011:

- New helper methods `rule()` and `isTerm()` described in Appendix B.

Changes from version of July 21, 2011:

- New parsing expressions: `^[`s`]`, e_1`*+`e_2, and e_1`++`e_2 described in Section 2.
- The use of `++` illustrated in Section 17 on Example 10.
- New helper method `where()` described in Appendix B.
- The use of `where()` illustrated in Section 13 on Example 7.
- Updated grammar in Appendix A.

1 Introduction: recursive-descent parsers with backtracking

Parsing Expression Grammar (PEG), introduced by Ford in [5–7], is a new way to specify a recursive-descent parser with limited backtracking.

Recursive-descent parsers have been around for a while. Already in 1961, Lucas [12] suggested the use of recursive procedures that reflect the syntax of the language being parsed. This close connection between the code and the grammar is the great advantage of recursive-descent parsers. It makes the parser easy to maintain and modify.

A recursive-descent parser is simple to construct from a classical context-free grammar if the grammar has the so-called $LL(1)$ property; it means that the parser can always decide what to do by looking at the next input character. However, forcing the language into the $LL(1)$ mold can make the grammar – and the parser – unreadable.

The $LL(1)$ restriction can be circumvented by the use of backtracking. Backtracking means that the parser proceeds by trial and error: goes back and tries another alternative if it took a wrong decision. However, an exhaustive search of all alternatives may require an exponential time. A reasonable compromise is limited backtracking, also called "fast-back" in [10].

Limited backtracking was adopted in at least two of the early top-down designs: the Atlas Compiler Compiler of Brooker and Morris [4, 18], and TMG (the TransMoGrifier) of McClure [13]. The syntax specification used in TMG was later formalized and analyzed by Birman and Ullman [2, 3]. It appears in [1] as "Top-Down Parsing Language" (TDPL) and "Generalized TDPL" (GTDPL). Parsing Expression Grammar is a development of this latter.

The name "Grammar" may be confusing when applied to what is essentially a top-down parsing language, but it was introduced in [7] as a new, recognition-based, method of defining syntax. It also has an appearance similar to grammars in the Extended Backus-Naur Form (EBNF) – although this similarity can often be misleading.

Parsers defined by PEG do not require a separate "lexer" or "scanner". Together with the lifting of the $LL(1)$ restriction, this gives a very convenient tool when we need an ad-hoc parser for some application.

Even the limited backtracking may require a lot of time. In [5, 6], PEG was introduced together with a technique called *packrat parsing*. Packrat parsing handles backtracking by extensive *memoization*: storing all results of parsing procedures[1]. It guarantees linear parsing time at a large memory cost. There exists a complete parser generator named *Rats!* [8, 9] that produces packrat parsers from PEG.

Excessive backtracking does not matter in small interactive applications where the input is short and performance not critical. Moreover, the usual programming languages do not require much backtracking. Experiments reported in [15, 16] demonstrated a moderate backtracking activity in PEG parsers for programming languages Java 1.5 and C.

Mouse is a development of parser generator used for these experiments. It translates PEG into a set of recursive procedures that closely follow the grammar. Unlike *Rats!*, *Mouse* does not produce a packrat parser. Optionally, it can offer a small amount of memoization using the technique described in [15]. Both *Mouse* and the resulting parser are written in Java. An integral feature of *Mouse* is the mechanism for specifying semantics (also in Java).

After a short presentation of PEG in the following section, the rest of the paper has the form of a tutorial, introducing the reader to *Mouse* by hands-on experience.

[1] "Packrat" comes from *pack rat* – a small rodent (*Neotoma cinerea*) known for hoarding unnecessary items; also a person that does the same. "Memoization", introduced in [14], is the technique of reusing stored results of function calls instead of recomputing them.

2 Parsing Expression Grammar

2.1 Parsing expressions

Parsing expressions are instructions for parsing strings, written in a special language. You can think of a parsing expression as shorthand for a procedure that carries out such instruction. The expressions can invoke each other recursively, thus forming together a recursive-descent parser.

In general, parsing expression is applied to a character string (the "input" to be parsed) at a position indicated by some "cursor". It tries to recognize a portion of the string ahead of the cursor. Usually, it "consumes" the recognized portion by advancing the cursor and indicates "success"; if it fails at the recognition, it indicates "failure" and does not consume anything (does not advance the cursor).

The following five kinds of parsing expressions work directly on the input string:

`"`s`"`
: where s is a nonempty character string. If the text ahead starts with the string s, consume that string and indicate success. Otherwise indicate failure.

`[`s`]`
: where s is a nonempty character string. If the text ahead starts with a character appearing in s, consume that character and indicate success. Otherwise indicate failure.

`^[`s`]`
: where s is a nonempty character string. If the text ahead starts with a character *not* appearing in s, consume that character and indicate success. Otherwise indicate failure.

`[`c_1`-`c_2`]`
: where c_1, c_2 are two characters. If the text ahead starts with a character from the range c_1 through c_2, consume that character and indicate success. Otherwise indicate failure.

`_`
: where `_` is the underscore character. If there is a character ahead, consume it and indicate success. Otherwise (that is, at the end of input) indicate failure.

These expressions are analogous to terminals of a classical context-free grammar, and are in the following referred to as "terminals". The remaining kinds of parsing expressions invoke other expressions to do their job:

e`?`
: Invoke the expression e and indicate success whether it succeeded or not.

e`*`
: Invoke the expression e repeatedly as long as it succeeds.
Indicate success even if it did not succeed a single time.

e`+`
: Invoke the expression e repeatedly as long as it succeeds.
Indicate success if e succeeded at least once. Otherwise indicate failure.

`&`e
: This is a lookahead operation ("syntactic predicate"): invoke the expression e and then reset the cursor as it was before the invocation of e (do not consume input).
Indicate success if e succeeded or failure if e failed.

`!`e
: This is a lookahead operation ("syntactic predicate"): invoke the expression e and then reset the cursor as it was before the invocation of e (do not consume input).
Indicate success if e failed or failure if e succeeded.

$e_1 \ldots e_n$
: Invoke expressions $e_1, \ldots, e_n$, in this order, as long as they succeed. Indicate success if all succeeded. Otherwise reset cursor as it was before the invocation of e_1 (do not consume input) and indicate failure.

e_1 `/` $\ldots$ `/` e_n
: Invoke expressions $e_1, \ldots, e_n$, in this order, until one of them succeeds. Indicate success if one of expressions succeeded. Otherwise indicate failure.

The following two expressions are shorthand forms of frequently used constructions. In addition to being easier to read, they have a more efficient implementaton.

e_1`*+` e_2
: A shorthand for `(!`$e_2\, e_1$`)*`e_2: iterate e_1 until e_2. More precisely: invoke e_2; if it fails, invoke e_1 and try e_2 again. Repeat this until e_2 succeeds. If any of the invocations of e_1 fails, reset cursor as it was at the start (do not consume input) and indicate failure.

e_1`++` e_2
: A shorthand for `(!`$e_2\, e_1$`)+`e_2, or `(!`$e_2\, e_1$`)(!`$e_2\, e_1$`)*`e_2: iterate e_1 at least once until e_2. Invoke e_2. If it succeeds, reset the cursor (do not consume input) and indicate failure. Otherwise invoke e_1 and try e_2 again. Repeat this until e_2 succeeds. If any of the invocations of e_1 fails, reset cursor as it was at the start (do not consume input) and indicate failure.

The following table summarizes all forms of parsing expressions.

expression	name	precedence
`"`s`"`	String Literal	5
`[`s`]`	Character Class	5
`^[`s`]`	Not Character Class	5
`[`c_1`-`c_2`]`	Character Range	5
`_`	Any Character	5
e`?`	Optional	4
e`*`	Iterate	4
e`+`	One or More	4
e_1`*+`e_2	Iterate Until	4
e_1`++`e_2	One or More Until	4
`&`e	And-Predicate	3
`!`e	Not-Predicate	3
$e_1 \ldots e_n$	Sequence	2
e_1 `/` $\ldots$ `/` e_n	Choice	1

The expressions $e, e_1, \ldots, e_n$ above can be specified either explicitly or by name (the way of naming expressions will be explained in a short while). An expression specified explicitly in another expression with the same or higher precedence must be enclosed in parentheses.

Backtracking takes place in the Sequence expression. If $e_1, \ldots, e_i$ in $e_1 \ldots e_i \ldots e_n$ succeed and consume some input, and then e_{i+1} fails, the cursor is moved back to where it was before trying e_1; the whole expression fails. If this expression was invoked from a Choice expression (and was not the last there), Choice has an opportunity to try another alternative on the same input.

However, the opportunities to try another alternative are limited: once e_i in the Choice expression $e_1/\ldots/e_i/\ldots/e_n$ succeeded, none of the alternatives $e_{i+1}, \ldots e_n$ will ever be tried on the same input, even if the parse fails later on.

2.2 The grammar

Parsing Expression Grammar is a list of one or more "rules" of the form:

name `=` *expr* `;`

where *expr* is a parsing expression, and *name* is a name given to it. The *name* is a string of one or more letters (`a-z`, `A-Z`) and/or digits, starting with a letter. White space is allowed everywhere except inside names. Comments starting with a double slash and extending to the end of a line are also allowed.

The order of the rules does not matter, except that the expression specified first is the "top expression", invoked at the start of the parser.

A specific grammar may look like this:

```
Sum    = Number ("+" Number)* !_ ;
Number = [0-9]+ ;
```

It consists of two named expressions: `Sum` and `Number`. They define a parser consisting of two procedures named `Sum` and `Number`. The parser starts by invoking `Sum`. The `Sum` invokes `Number`, and if this succeeds, repeatedly invokes `("+" Number)` as long as it succeeds. Finally, `Sum` invokes a sub-procedure for the predicate "`!_`", which succeeds only if it does not see any character ahead – that is, only at the end of input. The `Number` reads digits in the range from 0 through 9 as long as it succeeds, and is expected to find at least one such digit.

One can easily see that the parser accepts strings like "`2+2`", "`17+4711`", or "`2`"; in general, one or more integers separated by "`+`". You could guess this by reading the above as EBNF, which often leads you right. However, you should keep in mind that, in spite of its outward similarity, PEG *is not* EBNF.

If you are not familiar with PEG programming, you may get some insights into its problems and pitfalls from the author's papers [15–17].

It is quite obvious that the grammar you define must not be left-recursive, and the expression e in e`*` and e`+` must never succeed in consuming an empty string; otherwise, the parser could run into an infinite descent or an infinite loop. In [7] Ford defined a formal property *WF* (Well-Formed) that guarantees absence of these problems. *Mouse* computes this property and points out expressions that are not well-formed.

3 Getting started

For hands-on experience with *Mouse* you will need the executables and examples.
All materials, including this manual, are packaged in the `TAR` file `Mouse-1.5.1.tar.gz`, available from `http://sourceforge.net/projects/mousepeg`. Download and unpack this file to obtain directory `Mouse-1.5.1`. The directory contains, among other items, `JAR` file `Mouse-1.5.1.jar` and sub-directory `examples`. Add a path to that `JAR` in the `CLASSPATH` system variable.

The `examples` directory contains ten directories named `example1` through `example10` that you will use in your exercise.

All the examples use unnamed packages and assume that you do all the compilation and execution in a current work directory. You may create such directory specifically for this purpose; it is in the following called `work`. You need, of course, to have "current work directory" in the `CLASSPATH`.

4 The first steps

We start with the sample grammar from Section 2. You find it as the file `myGrammar.txt` in `example1`. Copy it to the `work` directory, and enter at the command prompt:

```
java mouse.Generate -G myGrammar.txt -P myParser
```

This should produce the answer:

```
2 rules
1 unnamed
3 terminals
```

and generate in the `work` directory a file `myParser.java` containing class `myParser`. This is your parser.

If you open the file in a text editor, you will see that the parsing expressions `Sum` and `Number` have been transcribed into parsing procedures in the form of methods in the class `myParser`. The unnamed inner expression `("+" Number)` has been transcribed into the method `Sum_0`:

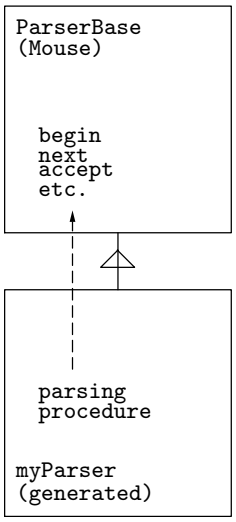
```
//======================================================================
//  Sum = Number ("+" Number)* !_ ;
//======================================================================
private boolean Sum()
  {
    begin("Sum");
    if (!Number()) return reject();
    while (Sum_0());
    if (!aheadNot()) return reject();
    return accept();
  }

//----------------------------------------------------------------------
//  Sum_0 = "+" Number
//----------------------------------------------------------------------
private boolean Sum_0()
  {
    begin("");
    if (!next('+')) return rejectInner();
    if (!Number()) return rejectInner();
    return acceptInner();
  }

//======================================================================
//  Number = [0-9]+ ;
//======================================================================
private boolean Number()
  {
    begin("Number");
    if (!nextIn('0','9')) return reject();
    while (nextIn('0','9'));
    return accept();
  }
```

The methods `begin()`, `accept()`, `reject()`, `next()`, etc. are standard services provided by *Mouse*. The first three maintain, behind the scenes, the information needed to define semantics; it will be discussed in the next section. As you may guess, `accept()` returns *true* and `reject()` returns *false*. The methods `next()` and `nextIn()` implement terminals. For example, `next('+')` implements the parsing expression `"+"`: if the next input character is `+`, consumes it and returns *true*, otherwise returns *false*.

As you can see by inspecting the file, class `myParser` is defined as a subclass of `mouse.runtime.ParserBase` provided by *Mouse*. The service methods are inherited from that superclass. The structure of your parser is thus as follows:



Compile your parser by typing:

```
javac myParser.java
```

To try the parser, type:

```
java mouse.TryParser -P myParser
```

This invokes a *Mouse* tool for running the parser. It responds with "`> `" and waits for input. If you enter a correct line, it prints another "`> `" to give you another chance. If your entry is incorrect, you get an error message before another "`> `". You exit by pressing Enter just after "`> `".

The session may look like this:

```
java mouse.TryParser -P myParser
> 1+2+3
> 17+4711
> 2#2
After '2': expected [0-9] or '+' or end of text
>
```

5 Adding semantics

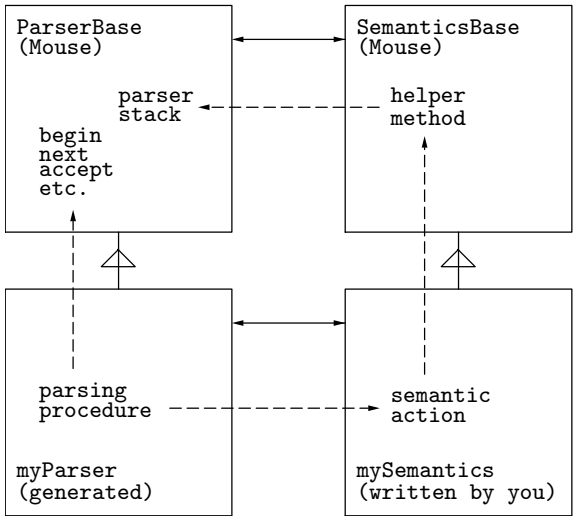
This hasn't been very exciting. You certainly want the parser to do something more than checking the syntax. For example, to actually compute the sum. For this purpose, you attach "semantic actions" to rules of the grammar.

As you have seen, *Mouse* transcribes each rule of the grammar into a parsing procedure in the form of a Java method. A semantic action is a Java method, written by you, that is called by a successful parsing procedure as the last thing before it exits. You may use this method to process the text consumed by the procedure.

In order to compute the sums, you may define a semantic action for `Number` that will convert the digits consumed by `Number` into an integer. You may then define a semantic action for `Sum` that will receive these integers, add them, and print the result.

In most parser generators, the code for semantic actions is inserted directly into the grammar. *Mouse* is a departure from that tradition. You provide semantic actions as methods in a separate class with any name of your choice, for example `"mySemantics"`. It must be a subclass of `mouse.runtime.SemanticsBase`

provided by *Mouse*. The superclass `SemanticsBase` contains "helper methods" that give you access to results of the parsing. The structure of your parser together with semantic actions is as follows:



All four classes are instantiated by a constructor that is included in the generated parser. The constructor establishes references between the instantiated objects.

In the grammar, you indicate the presence of a semantic action by inserting its name in curly brackets at the end of a rule:

```
Sum    = Number ("+" Number)* !_ {sum} ;
Number = [0-9]+ {number} ;
```

This tells that `Sum` will have a semantic action `sum()`, and `Number` will have a semantic action `number()`. You find this grammar as `myGrammar.txt` in `example2`. Copy it into the `work` directory (replacing the old one), and type at the command prompt:

```
java mouse.Generate -G myGrammar.txt -P myParser -S mySemantics
```

This generates a new file `myParser.java`. By inspecting this file, you can see that parsing procedures `Sum()` and `Number()` contain calls to their semantic actions, as shown below. The qualifier `sem` is a reference to your semantics class `mySemantics`; you had to supply its name in order to generate this reference.

```
//=====================================================================
//  Sum = Number ("+" Number)* !_ {sum} ;
//=====================================================================
private boolean Sum()
  {
    begin("Sum");
    if (!Number()) return reject();
    while (Sum_0());
    if (!aheadNot()) return reject();
    sem.sum();                          // semantic action
    return accept();
  }
```

```
  //=========================================================================
  //  Number = [0-9]+ {number} ;
  //=========================================================================
  private boolean Number()
    {
      begin("Number");
      if (!nextIn('0','9')) return reject();
      while (nextIn('0','9'));
      sem.number();                      // semantic action
      return accept();
```

Your semantics class is going to appear like this:

```
class mySemantics extends mouse.runtime.SemanticsBase
{
  //-------------------------------------------------------------------
  //  Sum = Number ("+" Number)* !_
  //-------------------------------------------------------------------
  void sum()
    {...}

  //-------------------------------------------------------------------
  //  Number = [0-9]+
  //-------------------------------------------------------------------
  void number()
    {...}
```

If you wish, you may generate a file containing the above skeleton at the same time as you generated your parser, by specifying the option `-s` on your command:

```
java mouse.Generate -G myGrammar.txt -P myParser -S mySemantics -s
```

It replies:

```
2 rules
1 unnamed
3 terminals
2 semantic procedures
```

indicating that it found two semantic actions, and produces the skeleton file `mySemantics.java` in the `work` directory, in addition to the parser file `myParser.java`.

It remains now to write the two methods. In order to do this, you need to know how to access the result of parsing.

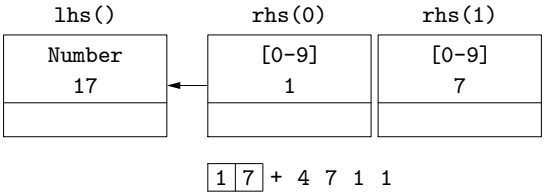
Behind the scenes, hidden in the service methods, each call to a parsing procedure, or to a service that implements a terminal, returns a `Phrase` object. The object is an instance of class `Phrase`, an internal class of `ParserBase`. It describes the portion of input consumed by the call – the consumed "phrase".

Still behind the scenes, each parsing procedure collects the `Phrase` objects returned by the service methods and procedures that it calls. At the end, it uses information from them to construct its own resulting `Phrase`. The semantic action is called just after this latter has been constructed, and can access it, together with the `Phrase`s representing the partial results.

As an illustration, we are going to describe what happens when your parser processes the input `"17+4711"`.

The process starts by calling `Sum()`, which immediately calls `Number()`. `Number()`, in turn, makes two calls to `nextIn()` that consume, respectively, the digits `"1"` and `"7"`, and return one `Phrase` object each. `Number()` (or, more precisely, the service methods called in the process) uses information from these objects to construct a new `Phrase` object that represents the text `"17"` thus consumed. This is going to be the result of `Number()`.

All three objects are accessible to the semantic action `number()`, and can be visualized as follows:



Each box represents one `Phrase` object. The text highest in the box identifies the parsing procedure or the service that constructed it. The text below is the consumed portion of input.

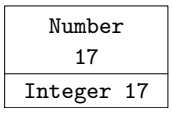
In the semantic action, you obtain references to the three objects by calls to helper methods shown above the boxes. They should be read as "left-hand side", "right-hand side element 0", and "right-hand side element 1": they correspond to the left- and right-hand sides of the rule `"Number = [0-9]+"`.

According to the initial plan, `number()` is expected to convert the text `"17"` consumed by `Number()` into the integer 17. This could be done by obtaining the digits represented by `rhs(0)` and `rhs(1)` and processing them to obtain the integer. However, the integer can be obtained directly from the text represented by `lhs()`, by using a method from the Java class `Integer`. To obtain the text represented by a `Phrase` object, you apply the helper method `text()` to the object. The integer 17 is thus obtained as `Integer.valueOf(lhs().text())`.

Each `Phrase` object has a field of type `Object` where you can insert the "semantic value" of the represented phrase. It is shown here as an empty slot at the bottom of each box. You can insert there the integer 17 as semantic value of the result. You insert semantic value into a `Phrase` object by applying the helper method `put(...)` to that object, so your `number()` can be written as follows:

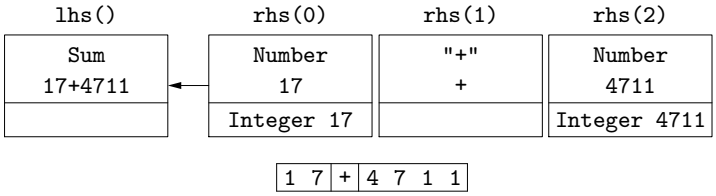
```
//-------------------------------------------------------------------
//  Number = [0-9]+
//-------------------------------------------------------------------
void number()
  { lhs().put(Integer.valueOf(lhs().text())); }
```

After completed semantic action, `Number()` returns to `Sum()`, delivering this object as its result:



After receiving control back, `Sum()` proceeds, via `Sum_0()`, to call `next()` and `Number()` that consume, respectively, `"+"` and `"4711"`, and return their resulting `Phrase` objects. `Sum()` uses information from them to construct a new `Phrase` object that represents the text `"17+4711"` thus consumed.

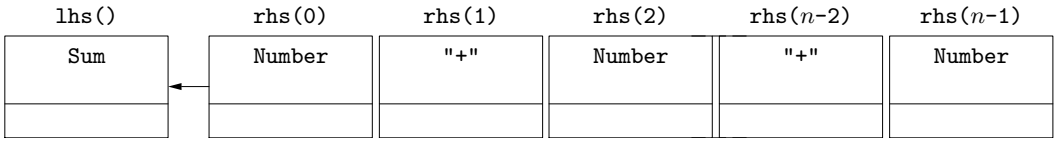
All four objects are accessible to the semantic action `sum()`, and can be visualized as follows:



Both `Number()` results carry as semantic values the integers computed by `number()`. To obtain the semantic value of a `Phrase` you apply the helper method `get()` to it. You can now easily compute the sum as follows (you need the casts because `get()` returns an `Object`):

```
(Integer)rhs(0).get() + (Integer)rhs(2).get().
```

However, in the general case you may have any number $p \geq 0$ of `("+" Number)` pairs resulting from p successful calls to `Sum_0()`. Your `sum()` must thus handle the following general situation with $n = 2p+1$ `Phrase`s on the right-hand side:



To obtain the number n you use the helper method `rhsSize()`. Your method `sum()` may thus look like this:

```
//--------------------------------------------------------------------
//  Sum = Number "+" Number ... "+" Number !_
//          0     1    2        n-2   n-1
//--------------------------------------------------------------------
void sum()
  {
    int s = 0;
    for (int i=0;i<rhsSize();i+=2)
      s += (Integer)rhs(i).get();
    System.out.println(s);
  }
```

You find the complete semantics class, with the methods `number()` and `sum()` just described, as file `mySemantics.java` in `example2`; copy it to the `work` directory. To obtain the new parser, compile the semantics class and the parser class you have generated. You can try it in the same way as before. The session may look like this:

```
javac myParser.java
javac mySemantics.java
java mouse.TryParser -P myParser
> 17+4711
4728
> 12+2+3
17
>
```

6 Understanding the "right-hand side"

As can be seen from the above example, you need to know the exact appearance of the "right-hand side" in order to correctly write your semantic action. Your method must be prepared to handle a varying configuration that depends on the parsed text. To have this configuiration in front of you while writing the code, it is convenient to describe it by a comment as shown above. (The author has learned this style from his colleague Bertil Steinholtz, who used it to write very clear `yacc++` code.)

The objects appearing on the "right-hand side" correspond to the names and terminals appearing to the right of `"="` in the rule. You can find the possible configurations of the "right-hand side" like this:

> Take the expression to the right of `"="` and remove from it all predicates, that is, sub-expressions of the form $\&e$ and $!e$. Replace each `"/"` by `"|"`, `"*+"` by `"*"`, and `"++"` by `"+"`. The result is a regular expression on an alphabet consisting of names and terminals treated as single letters. The possible configurations of the "right-hand side" are exactly the strings defined by this regular expression.

As an example, the regular expression obtained for `Sum` is:

```
Number ("+" Number)*
```

which indeed defines these strings of symbols `Number` and `"+"`:

```
Number "+" Number ... "+" Number
  0     1    2        n-2  n-1
```

where $n = 2p + 1$ for $p = 0, 1, 2, \ldots$.

7 Getting more realistic

Our example so far is not very realistic. Any decent parser will allow white space in the input. This is simply achieved in PEG by defining `Space` as a sequence of zero or more blanks, and inserting `Space` in strategic places. We are going to do this, and on this occasion add two features: an optional minus sign in front of the first `Number`, and an alternative minus between the `Number`s:

```
Sum     = Space Sign Number (AddOp Number)* !_ {sum} ;
Number  = Digits Space {number} ;
Sign    = ("-" Space)? ;
AddOp   = [-+] Space ;
Digits  = [0-9]+ ;
Space   = " "* ;
```

The semantic action `number()` will now see this configuration of the "right-hand side":

```
Digits Space
  0      1
```

The text to be converted is now represented by `rhs(0)` rather than `lhs()`, so `number()` appears as follows:

```
//------------------------------------------------------------------
//  Number = Digits Space
//             0      1
//------------------------------------------------------------------
void number()
  { lhs().put(Integer.valueOf(rhs(0).text())); }
```

Note that we could have defined `Number` as `[0-9]+ Space`, but this would present `number()` with a "right-hand side" of variable length, and give it a job of collecting all the digits. It is much easier to let the parser do the job.

The semantic action `sum()` will see this configuration of the "right-hand side":

```
Space Sign Number AddOp Number ... AddOp Number
  0     1    2      3     4          n-2   n-1
```

where $n = 2p + 3$ for $p = 0, 1, 2, \ldots$.

The text for `Sign` is either an empty string or a minus followed by zero or more blanks. You can check which is the case with the helper method `isEmpty()`. The text for `AddOp` is either a plus or a minus followed by zero or more blanks. You can obtain the first character of this text with the helper method `charAt()`. This gives the following new version of `sum()`:

```
//------------------------------------------------------------------
//  Sum = Space Sign Number AddOp Number ... AddOp Number
//          0     1    2      3     4          n-2   n-1
//------------------------------------------------------------------
void sum()
  {
    int n = rhsSize();
    int s = (Integer)rhs(2).get();
    if (!rhs(1).isEmpty()) s = -s;
    for (int i=4;i<n;i+=2)
    {
      if (rhs(i-1).charAt(0)=='+')
        s += (Integer)rhs(i).get();
      else
        s -= (Integer)rhs(i).get();
    }
    System.out.println(s);
  }
```

Note again that we could have `("-" Space)?` instead of `Sign` in the definition of `Sum`. But, according to the rule given in Section 6, `sum()` would then have to handle two different configuration of the "right-hand side": one starting with (`Space "-" Space Number`...) and one starting with (`Space Number`...). In our definition of `Sum`, `Sign` is always there, even if the underlying (`"-" Space`) failed. Note that `Space` is also present even if it did not consume any blanks.

The above illustrates the fact that you have to exercise a lot of care when defining rules that will have semantic actions. Introducing `Digits` and `Sign` as separate named expressions made the "right-hand side" in each case easier to handle. For reasons that should now be obvious, you are not recommended to have repetitions or optional expressions inside repetitions. A clever programming of semantic action can always handle them, but the job is much better done by the parser.

You find the grammar and the semantics class for this example as `myGrammar.txt` and `mySemantics.java` in `example3`. Copy them to the `work` directory (replacing the old ones), generate the parser, and compile it as before. A session with the new parser may look like this:

```
java mouse.TryParser -P myParser
>   17 + 4711
4728
> -1 + 2 - 3
-2
>
```

To close this section, note that you implemented here some functions traditionally delegated to a separate "lexer": conversion of integers and ignoring white space. If you wanted, you could easily include newlines and comments in `Space`.

8 Let's go floating

To illustrate some more features of *Mouse*, we shall add floating-point numbers to your primitive calculator. The calculations will now be done on numbers in `double` form. The only thing needed in the grammar is the possibility to specify such numbers. We decide for a format with one or more digits after the decimal point, and optional digits before: `Digits? "." Digits`. We are going to keep the integer format as an alternative, which means defining `Number` as

```
Number = Digits? "." Digits Space / Digits Space ;
```

The two alternatives produce two quite different configurations of the "right-hand side". The semantic action would have to begin by identifying them. But, this has already been done by the parser. To exploit that, you specify a separate semantic action for each alternative in a rule that defines choice expression. It means modifying your grammar as follows:

```
Sum     = Space Sign Number (AddOp Number)* !_ {sum} ;
Number  = Digits? "." Digits Space {fraction}
        / Digits Space {integer} ;
Sign    = ("-" Space)? ;
AddOp   = [-+] Space ;
Digits  = [0-9]+ ;
Space   = " "* ;
```

You specify here two semantic actions, `fraction()` and `integer()`, for the two alternatives of `Number`.

The first of them is invoked to handle input of the form `Digits? "." Digits Space`. According to the rule given in Section 6, it can result in two possible configurations of the "right-hand side":

```
Digits "." Digits Space       or       "." Digits Space
   0    1     2     3                    0     1     2
```

You need to convert the text that represents the floating-point number into a `Double` object. You can obtain this text by using the helper method `rhsText(`i, j`)`. The method returns the text identified by the right-hand side elements i through $j-1$. As you can easily see, `rhsText(0,rhsSize()-1)` will do the job for each of the two configurations, so `fraction()` can be coded as follows:

```
//-------------------------------------------------------------------
//  Number = Digits? "."  Digits Space
//              0    1(0)  2(1)   3(2)
//-------------------------------------------------------------------
void fraction()
  { lhs().put(Double.valueOf(rhsText(0,rhsSize()-1))); }
```

As computation will now be done in floating point, `integer()` should also return a `Double`:

```
//-------------------------------------------------------------------
//  Number = Digits Space
//              0     1
//-------------------------------------------------------------------
void integer()
  { lhs().put(Double.valueOf(rhs(0).text())); }
```

The semantic action for `Sum` remains unchanged, except that `s` becomes `double` and the casts are to `Double` instead of `Integer`.

You may copy the new grammar and semantics from `example4`, generate new parser (replacing the old), and compile both. A session with the new parser may look like this:

```
java mouse.TryParser -P myParser
> -123 + 456
  333.0
> 1.23 - 4.56
  -3.3299999999999996
>
```

(Who said floating-point arithmetic is exact?)

9 What about backtracking?

As you can easily see, `Number` in the last grammar does not satisfy the classical $LL(1)$ condition: seeing a digit as the first character, you do not know which alternative it belongs to. Presented with "`123`", the parser will start with the first alternative of `Number` ("fraction") and process "`123`" as its `Digits`. Not finding the decimal point, the parser will backtrack and try the other alternative ("integer"), again processing the same input as `Digits`. This re-processing of input is the price for not bothering to make the grammar $LL(1)$.

The loss of performance caused by reprocessing is uninteresting in this particular application. However, circumventing $LL(1)$ in this way may cause a more serious problem. The reason is the limited backtracking of PEG. As indicated in Section 2, if the parser successfully accepts the first alternative of e_1/e_2 and fails later on, it will never return to try e_2. As a result, some part of the language of e_2 may never be recognized. This is a known effect; in [17] it was called "language hiding": a greedy e_1 hides a part, or all, of the language otherwise recognized by e_2.

Just imagine for a while that the two alternatives in `Number` are reversed. Presented with "`123.45`", `Number` consumes "`123`" as an "integer" and returns to `Sum`, reporting success. The `Sum` finds a point instead of `AddOp`, and terminates prematurely. The other alternative of `Number` is never tried. All fractional numbers starting with `Digits` are hidden by a greedy "integer".

Sylvain Schmitz [19] formulated the following sufficient condition for the absence of language hiding:

$$\mathcal{L}(e_1) \cap \mathrm{Pref}(\mathcal{L}(e_2)\mathcal{L}(\mathrm{follow}(e_1/e_2))) = \varnothing\,,$$

where $\mathcal{L}(e)$ is the language of e, follow(e) is any sequence of expressions that can be called after e, and $\mathrm{Pref}(\mathcal{L})$ is the set of all prefixes of strings in $\mathcal{L}$.

An automatic checking of this condition is complex, and perhaps even impossible in general case. *Mouse* does not attempt it. However, you can still use the formula to verify by inspection that the choice in `Number` starting with "fraction" is safe: the "integer" is never followed by a decimal point.

To watch the backtracking activity of your parser, you may generate an instrumented version of it. To do this, type these commands:

```
java mouse.Generate -G myGrammar.txt -P myParser -S mySemantics -T
javac myParser.java
```

The option `-T` instructs the Generator to construct a "test version". (You may choose another name for this version if you wish, the semantics class remains the same.) Invoking the test version with `mouse.TryParser` produces the same results as before. In order to exploit the instrumentation, you have to use `mouse.TestParser`; the session may look like this:

```
java mouse.TestParser -P myParser
> 123 + 4567
  4690.0

line 1: 10 bytes.
51 calls: 34 ok, 15 failed, 2 backtracked.
11 rescanned.
backtrack length: max 4, average 3.5.
>
```

This output tells you that to process your input "`123 + 4567`", the parser executed 51 calls to parsing procedures, of which 34 succeeded, 15 failed, and two backtracked. (We treat here the services that implement terminals as parsing procedures.) As expected, the parser backtracked 3 characters on the first `Number` and 4 on the second, so the maximum backtrack length was 4 and the average backtrack length was 3.5. You can also see that 11 of the procedure calls were "re-scans": the same procedure called again at the same input position. You can get more detail by specifying the option `-d`:

```
java mouse.TestParser -P myParser -d
> 123 + 4567
  4690.0

line 1: 10 bytes.
51 calls: 34 ok, 15 failed, 2 backtracked.
11 rescanned.
backtrack length: max 4, average 3.5.

Backtracking, rescan, reuse:
procedure        ok  fail  back  resc reuse totbk maxbk at
------------- ----- ----- ----- ----- ----- ----- ----- --
Digits            4     0     0     2     0     0     0
Number_0          0     0     2     0     0     7     4 After '123 + '
[0-9]            14     4     0     9     0     0     0

>
```

You see here statistics for individual procedures that were involved in backtracking and rescanning. `Number_0` is the internal procedure for the first alternative of `Number`. As you can guess, "`totbk`" stands for total backtrack length and "`maxbk`" for length of the longest backtrack; "`at`" tells where this longest backtrack occurred. The meaning of "`reuse`" will be clear in a short while.

10 A mouse, not a pack rat

As mentioned in the introduction, *Mouse* can provide a small degree of memoization. You can exercise this possibility by specifying the option `-m` to `TestParser`. You follow it by a digit from 1 to 9, indicating how many most recent results you want to keep. For example, your session from the previous section, with procedures caching one result, will appear like this:

```
java mouse.TestParser -PmyParser -d -m1
> 123 + 4567
  4690.0

line 1: 10 bytes.
42 calls: 25 ok, 13 failed, 2 backtracked.
0 rescanned, 2 reused.
backtrack length: max 4, average 3.5.

Backtracking, rescan, reuse:

procedure        ok  fail  back  resc reuse totbk maxbk at
------------ ----- ----- ----- ----- ----- ----- ----- --
Digits            2     0     0     0     2     0     0
Number_0          0     0     2     0     0     7     4 After '123 + '

>
```

You can see here that the parser reused the cached result of `Digits` on two occasions, thus eliminating the unnecessary rescanning by `[0-9]`.

If you decide you want memoization, you can generate a version of the parser that allows it, without all the instrumentation overhead. You do it by specifying the option `-M` on `mouse.Generate`. You specify then the amount of memoization by means of `-m` option in the same way as above.

When deciding whether you want memoization or not, you should consider the fact that it introduces some overhead. It may cost more in performance than some moderate rescanning. In the current version of *Mouse*, memoization is not applied to the services for terminals because the overhead of memoization was felt to be larger than the job of rescanning a terminal.

11 Full arithmetic

As the next development, we suggest adding multiplication, division and parentheses:

```
Input   = Space Sum !_ {result} ;
Sum     = Sign Product (AddOp Product)* {sum} ;
Product = Factor (MultOp Factor)* {product} ;
Factor  = Digits? "." Digits Space {fraction}
        / Digits Space {integer}
        / Lparen Sum Rparen {unwrap} ;
Sign    = ("-" Space)? ;
AddOp   = [-+] Space ;
MultOp  = [*/]? Space ;
Lparen  = "(" Space ;
Rparen  = ")" Space ;
Digits  = [0-9]+ ;
Space   = " "* ;
```

We have replaced here `Number` by `Factor` that can be a fraction, an integer, or a sum in parentheses.

We have added `Product` with multiplication and division to ensure, in a syntactical way, the priority of these operations over addition and subtraction. The omitted operator in `MultOp` is intended to mean multiplication.

As `Sum` can now appear in `Factor`, we had to add `Input` as new top level, moving there the initial `Space` and the check for end of input.

The semantic action `sum()` cannot now print its result; instead, the result will be inserted as semantic value into the `Sum`'s result, to be retrieved by the semantic action of `Input` or `Factor`:

```
//---------------------------------------------------------------------
//  Sum = Sign Product AddOp Product ... AddOp Product
//          0     1      2      3         n-2    n-1
//---------------------------------------------------------------------
void sum()
  {
    int n = rhsSize();
    double s = (Double)rhs(1).get();
    if (!rhs(0).isEmpty()) s = -s;
    for (int i=3;i<n;i+=2)
    {
      if (rhs(i-1).charAt(0)=='+')
        s += (Double)rhs(i).get();
      else
        s -= (Double)rhs(i).get();
    }
    lhs().put(new Double(s));
  }
```

This assumes, of course, that `Product` delivers its result as semantic value of its result. The semantic action `product()` follows the same pattern as `sum()`:

```
//---------------------------------------------------------------------
//  Product = Factor MultOp Factor ... MultOp Factor
//              0     1      2         n-2     n-1
//---------------------------------------------------------------------
void product()
  {
    int n = rhsSize();
    double s = (Double)rhs(0).get();
    for (int i=2;i<n;i+=2)
    {
      if (!rhs(i-1).isEmpty() && rhs(i-1).charAt(0)=='/')
        s /= (Double)rhs(i).get();
      else
        s *= (Double)rhs(i).get();
    }
    lhs().put(new Double(s));
  }
```

(`MultOp` means division if it is a nonempty string starting with with `"/"`; otherwise it is multiplication.)

Note that the precedence of multiplication and division over addition and subtraction is here defined by the syntax. However, the order of operations within each class (left to right) is *not* defined by the syntax, but by semantic actions.

The actions `fraction()` and `integer()` remain the same as in the preceding example. The function of `unwrap()` is to take the semantic value of `Sum` and deliver it as the semantic value of `Factor`:

```
//---------------------------------------------------------------------
//  Factor = Lparen Sum Rparen
//             0    1    2
//---------------------------------------------------------------------
void unwrap()
  { lhs().put(rhs(1).get()); }
```

No casts are needed. It remains to print the result:

```
//---------------------------------------------------------------------
//  Input = Space Sum !_
//            0    1
//---------------------------------------------------------------------
void result()
  { System.out.println((Double)rhs(1).get()); }
```

The modified grammar and semantics are available in `example5`; you may copy them into the `work` directory, then generate and compile the parser as before. The session may now look like this:

```
java mouse.TryParser -P myParser
> 2(1/10)
  0.2
> (1+2)*(3-4)
  -3.0
> 1.23(-4+.56)
  -4.2312
>
```

Notice that the multiplication operator can, in particular, be the empty string. This would cause an ambiguity in the classical context-free grammar, where `"234"` could be parsed as meaning `2*3*4`, `23*4`, or `2*34`. However, there is no place for ambiguity in the parser specified by PEG: it will always read the string as integer `234`. Which is, by the way, exactly how a human reader sees it.

12 Want a tree?

Many existing parsers deliver their result in the form of a syntax tree, showing how the input was parsed according to the grammar rules. The preceding examples show that much of this information may be uninteresting if the grammar is specified at the level of input characters rather than "tokens" produced by a "lexer". For example, the structure of our fractional number in terms of individual characters is not interesting as long as we only want the complete string. For this reason, *Mouse* does not provide automatic construction of syntax trees.

If you wish, you may still construct syntax tree at the desired level of detail with the help of semantic actions. In the following, we show how the preceding example can be modified to produce an operand-operator tree instead of calculating the result. Such tree can be then further processed (for example, to generate executable code). All you need to construct the tree is write another semantics class, without changing the parser.

Of course, you also need to define elements of the tree to be constructed. You find their definitions in the file `Node.java` in `example6`. It defines an abstract class `Node` with two subclasses: `Node.Num` and `Node.Op`, to represent, respectively, a number and a binary operator. In order to simplify the example, we do not provide separate subclasses for different kinds of numbers and operators. A number is stored in `Node.Num` as the text string retrieved from the input; an operator is stored in `Node.Op` as one character.

The semantic values will now be partial trees, represented by their top nodes. The semantic actions will appear like this:

```
//------------------------------------------------------------------
//  Factor = Digits? "."  Digits Space
//              0    1(0)  2(1)   3(2)
//------------------------------------------------------------------
void fraction()
  { lhs().put(new Node.Num(rhsText(0,rhsSize()-1))); }
```

```
//------------------------------------------------------------------
//  Sum = Sign Product AddOp Product ... AddOp Product
//          0     1      2      3         n-2    n-1
//------------------------------------------------------------------
void sum()
  {
    int n = rhsSize();
    Node N = (Node)rhs(1).get();
    if (!rhs(0).isEmpty())
      N = new Node.Op(new Node.Num("0"),'-',N);
    for (int i=3;i<n;i+=2)
      N = new Node.Op(N,rhs(i-1).charAt(0),(Node)rhs(i).get());
    lhs().put(N);
  }
```

(Minus as `Sign` is represented by subtraction from 0.)

You find complete semantics in `example6`. As you can see there, the final action, `result()`, saves the top node of the resulting tree in the variable `tree`, local to `mySemantics`. You cannot use the standard

tool `mouse.TryParser` to obtain the tree thus stored; you have to code invocation of the parser on your own. It will be an occasion to learn how this is done. The code is shown below.

```
import mouse.runtime.SourceString;
import java.io.BufferedReader;
import java.io.InputStreamReader;

class Try
{
  public static void main(String argv[])
    throws Exception
    {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      myParser parser = new myParser();            // Instantiate Parser+Semantics
      while (true)
      {
        System.out.print("> ");                    // Print prompt
        String line = in.readLine();               // Read line
        if (line.length()==0) return;              // Quit on empty line
        SourceString src = new SourceString(line); // Wrap up the line
        boolean ok = parser.parse(src);            // Apply Parser to it
        if (ok)                                    // If succeeded:
        {
          mySemantics sem = parser.semantics();    // Access Semantics
          System.out.println(sem.tree.show());     // Get the tree and show it
        }
      }
    }
}
```

Most of it is self-explanatory. The class `SourceString` supplied by *Mouse* is a wrapper for the parser's input. It contains methods needed to access the string and to print error position in diagnostics. After instantiating the parser, you apply it to the wrapped input by calling the method `parse()`. The method returns a `boolean` result indicating whether the parse was successful.

When the parser is instantiated, it also instantiates its semantics class. You can access that class by calling the method `semantics()` on the parser. You can then obtain the tree stored there. This is done in the last line that exploits the method `show()` of `Node` to print out the tree.

The above code is found in `example6` under the name `Try.java`. Copy the files `Node.java`, `mySemantics.java`, and `Try.java` into the `work` directory and compile them. You may now type `java Try` at the command prompt, and enter your input after "`> `". You will get the resulting tree as a nested expression where each operator node with its sub-nodes is enclosed by a pair of brackets `[ ]`. Your session may now look like this:

```
java Try
> 1+2+3
  [[1+2]+3]
> 2(1/10)
  [2*[1/10]]
> (1+2)*(3-4)
  [[1+2]*[3-4]]
> 1.23(-4+.56)
  [1.23*[[0-4]+.56]]
>
```

This is not the most elegant presentation of a tree, but the purpose is just to check what was constructed.

13 Calculator with memory

Back to the calculator from Section 11. We suggest one more improvement: the ability to store computed values for subsequent use. The idea is that a session may look like this:

```
java Calc
> n = 3/2
> n = (3/n + n)/2
> (3/n + n)/2
  1.7321428571428572
```

In the first line we store the number `3/2` under name "`n`". In the second, we replace it by `(3/n + n)/2`, computed using the stored value. Finally, we use the newest `n` to compute one more value and print it. (Is it just an accident that the result resembles $\sqrt{3}$?)

To achieve this kind of functionality, you can modify the grammar as follows:

```
Input   = Space (Store / Print) !_ ;
Store   = Name Space Equ Sum {store} ;
Print   = Sum {print} ;
Sum     = Sign Product (AddOp Product)* {sum} ;
Product = Factor (MultOp Factor)* {product} ;
Factor  = Digits? "." Digits Space {fraction}
        / Digits Space {integer}
        / Lparen Sum Rparen {unwrap}
        / Name Space {retrieve} ;
Sign    = ("-" Space)? ;
AddOp   = [-+] Space ;
MultOp  = [*/]? Space ;
Lparen  = "(" Space ;
Rparen  = ")" Space ;
Equ     = "=" Space ;
Digits  = [0-9]+ ;
Name    = [a-z]+ ;
Space   = " "* ;
```

We have here two alternative forms of `Input` to perform the two tasks, and one more way to obtain a `Factor`: by retrieving a stored result. The results will be stored in a hash table `dictionary` that resides in the semantics object.

The semantic actions for `Store` and `Print` are:

```
//------------------------------------------------------------------
//  Store = Name Space Equ Sum
//            0    1    2   3
//------------------------------------------------------------------
void store()
  { dictionary.put(rhs(0).text(),(Double)rhs(3).get()); }
```

```
//------------------------------------------------------------------
//  Print = Sum
//           0
//------------------------------------------------------------------
void print()
  { System.out.println((Double)rhs(0).get()); }
```

The stored result is retrieved by semantic action for the fourth alternative of `Factor`:

```
//--------------------------------------------------------------------
//  Factor = Name Space
//             0    1
//--------------------------------------------------------------------
void retrieve()
  {
    String name = rhs(0).text();
    Double d = dictionary.get(name);
    if (d==null)
    {
      d = (Double)Double.NaN;
      System.out.println
        (rhs(0).where(0) + ": '" + name + "' is not defined.");
    }
    lhs().put(d);
  }
```

You have to handle the case where the name is not in the dictionary. The example shown here returns in such a case a `NaN` (Not a Number) object. Using `NaN` in subsequent calculations produces inevitably a `NaN` result. A message printed to `System.out` explains where this `NaN` comes from. Note the call to `rhs(0).where(0)` that returns a text telling where to find the name in the input.

You find the modified grammar and semantics in `example7`. You find there also a file `Calc.java` containing invocation of the calculator. Copy these files to `work` directory, generate a new parser, and compile everything. You can now invoke the calculator as shown at the beginning of this section.

14 Get error handling right

Try the following with the calculator just constructed:

```
java Calc
> 2a
After '2': 'a' is not defined
NaN
> a = 12(
After 'a = 12': expected Space or Sum
> a
12.0
> 2a)
24.0
After '2a': expected [a-z] or Space or MultOp or Factor or AddOp or end of text
>
```

Note that the input `"a = 12("` resulted in failure, but response to the subsequent input `"a"` shows that `"a"` has been defined. The input `"2a)"` resulted again in failure, but the result of `"2a"` has been printed.

What happens?

In the first case, `Store` successfully consumed `"a = 12"`, and its action `store()` entered `"a"` into the dictionary with `12` as value. Then it returned success to `Input` which failed by finding `"("` instead of end of text, backtracked, and issued the error message. In the second case, `Print` successfully consumed `"2a"`, printed its result, and returned succes to `Input` wich failed by not finding end of text.

The moral of the story is that **you must not take any irreversible actions in a situation that can be reversed by backtracking**.

A simple solution in this case is by making the predicate `"!␣"` part of `Store` and `Print`:

```
Input   = Space (Store / Print) ;
Store   = Name Space Equ Sum !_ {store} ;
Print   = Sum !_ {print} ;
```

In this way, `store()` and `print()` are called only after the entire input has been successfully parsed.

You may wonder why the message after `"a = 12("` says nothing about the expected end of text.

A non-backtracking parser stops after failing to find an expected character in the input text, and this failure is reported as *the* syntax error. A backtracking parser may instead backtrack and fail several times. It terminates and reports failure when no more alternatives are left. The strategy used by *Mouse* is to report only the failure that occurred farthest down in the input. If several different attempts failed at the same point, all such failures are reported.

In the example, the message says that `Lparen` expected blanks after `"("`, and that the third alternative of `Factor` expected `Sum` at that place. The failure to find end of text occurred earlier in the input and is not mentioned.

In the case of `"2a)"`, several procedures failed immediately after `"2a"`. The processing did not advance beyond that point, so all these failures are reported.

You may feel that information about `Space` not finding a blank is uninteresting noise. To suppress it, you can define the following semantic action for `Space`:

```
//--------------------------------------------------------------------
//  Space = " "*
//--------------------------------------------------------------------
void space()
  { lhs().errClear(); }
```

When `space()` is invoked, the `lhs()` object seen by it contains the information that `Space()` ended by not finding another blank. The helper method `errClear()` erases all failure information from the `lhs()` object, so `Space` will never report any failure.

If you wish, you can make messages more readable by saying `"+ or -"` instead of `"AddOp"`. You specify such "diagnostic names" in pointed brackets at the end of a rule, like this:

```
AddOp   = [-+] Space  <+ or -> ;
MultOp  = [*/]? Space <* or /> ;
Lparen  = "(" Space   <(> ;
Rparen  = ")" Space   <)> ;
Equ     = "=" Space   <=> ;
```

A grammar and semantics with these modifications are found in `example8`. Copy them to the `work` directory, generate new parser (replacing the old), and compile both. A parser session may now appear like this:

```
java Calc
> 2a
After '2': 'a' is not defined
NaN
> a = 12(
At start: 'a' is not defined
After 'a = 12': expected Sum
> a
At start: 'a' is not defined
NaN
> 2a)
After '2': 'a' is not defined
After '2a': expected [a-z] or * or / or Factor or + or - or end of text
>
```

The message about `"a"` not being defined in response to `"a = 12("` is another example of an irreversible action in a situation reversed by backtracking. It appears because `Print` is attempted after the failure of `Store`, and issues the message before itself failing. Perhaps a better strategy should be invented for reporting undefined names, but it does not seem disturbing, and we leave it as it is. Note, however, that it would be entirely wrong if we reversed the order of `Store` and `Print`. In that case, a session would look like this:

```
java Calc
> a = 12
At start: 'a' is not defined
> a
12.0
>
```

Here, the parser applies `Print` first, reports `"a"` undefined, fails by finding `"="`, backtracks, and successfully executes `Store`.

15 Backtracking again

The grammar just constructed has another non-$LL(1)$ choice in addition to the choice between two kinds of number. It is the choice between `Store` and `Print` in `Input`: they can both start with `Name`.

If you enter something like "`lambda * 7`", the parser tries `Store` first and processes "`lambda `", expecting to find equal sign next. Finding "`*`" instead, the parser backtracks and tries `Print`. It eventually comes to process "`lambda `" via `Sum`, `Product`, and `Factor`. To watch this activity, you may generate test version of the parser using option `-T`, as described in Section 9 and try it. A possible result is shown below.

```
java mouse.TestParser -PmyParser -d
> lambda = 12

line 1: 11 bytes.
62 calls: 31 ok, 30 failed, 1 backtracked.
10 rescanned.
backtrack length: max 2, average 2.0.

Backtracking, rescan, reuse:

procedure        ok  fail  back  resc reuse totbk maxbk at
------------- ----- ----- ----- ----- ----- ----- ----- --
Digits            2     2     0     2     0     0     0
Space             5     0     0     1     0     0     0
Factor_0          0     1     1     0     0     2     2 After 'lambda = '
"."               0     2     0     1     0     0     0
[0-9]             4     4     0     5     0     0     0
" "               2     5     0     1     0     0     0

> lambda * 7
84.0

line 2: 10 bytes.
87 calls: 41 ok, 44 failed, 2 backtracked.
22 rescanned.
backtrack length: max 7, average 4.0.

Backtracking, rescan, reuse:

procedure        ok  fail  back  resc reuse totbk maxbk at
------------- ----- ----- ----- ----- ----- ----- ----- --
Store             0     0     1     0     0     7     7 At start
Digits            2     4     0     3     0     0     0
Name              2     1     0     1     0     0     0
Space             6     0     0     2     0     0     0
Factor_0          0     2     1     0     0     1     1 After 'lambda * '
"."               0     3     0     1     0     0     0
[0-9]             2     6     0     5     0     0     0
[a-z]            12     3     0     7     0     0     0
" "               3     6     0     3     0     0     0
>
```

This is quite a lot of rescanning; you may try to see the effects of specifying "`-m1`": the number of rescans is reduced to 2 for each input.

16 Input from file

In more serious applications, like a compiler, the parser has to read input from a file. We are going to show the way of doing this by an adaptation of the calculator developed above.

Just suppose you want to perform a sequence of calculations such as this:

```
n = 0
f = 1
e = 1
e
n = n + 1
f = f * n
e = e + 1/f
e
n = n + 1
f = f * n
e = e + 1/f
e
  ...
```

that will print the consecutive stages of computing the number e from its series expansion. You want the calculator to read this input from a file. All you need is to enlarge your syntax by defining the input as a sequence of `Line`s:

```
Input = [\n\r]* Line+ EOF ;
Line  = Space (Store / Print) ;
Store = Name Space Equ Sum EOL {store} ;
Print = Sum EOL {print} ;

 ...  as before  ...

EOL     = [\n\r]+ / EOF <end of line> ;
EOF     = !_ <end of file> ;
```

This grammar defines `EOL` to be either end of line or end of input. End of line is defined as one or more newlines and/or carriage returns, meaning that the syntax allows empty lines – really empty, without even space characters in them. (The `"\n"` and `"\r"` stand for newline and carriage return characters, respectively; see Section 18.1.) The `[\n\r]*` at the beginning permits the file to start with empty lines.

You do not need to change semantics.

Copy `myGrammar.txt` and sample file `e.txt` from `example9`, generate the parser and compile it in the usual way.

To run the parser on the sample file, type at command prompt:

```
java mouse.TryParser -P myParser -f e.txt
```

The option `-f e.txt` instructs the tool to take its input from the file `e.txt`. You should get this output:

```
1.0
2.0
2.5
2.6666666666666665
2.708333333333333
2.7166666666666663
2.7180555555555554
2.7182539682539684
2.71827876984127
2.7182815255731922
2.7182818011463845
e.txt: ok
```

The *Mouse* tool `TryParser` is provided to help you in developing the parser, but eventually you will need to invoke the parser from one of your own programs. The following is an example of how you can do it:

```
import mouse.runtime.SourceFile;
class FileCalc
{
  public static void main(String argv[])
    {
      myParser parser = new myParser();          // Instantiate Parser+Semantics
      SourceFile src = new SourceFile(argv[0]);  // Wrap the file
      if (!src.created()) return;                // Return if no file
      boolean ok = parser.parse(src);            // Apply parser to it
      System.out.println(ok? "ok":"failed");     // Tell if succeeded
    }
}
```

The class `SourceFile` supplied by *Mouse* is a wrapper that maps the file in memory and contains methods needed to access it and to print error position in diagnostics. You find the above code as `FileCalc.java` in `example9`. Copy it into the `work` directory and compile. You can then invoke the calculation by typing:

```
java FileCalc e.txt
```

To see the kind of diagnostics produced for a file, damage `e.txt` by, for example, duplicating one of the operators or changing `"n"` to `"x"` in an expression, and run the parser on the damaged file.

17 Error recovery

If you tried the parser on a faulty file, you have noticed that it stops after encountering the first faulty line. It perhaps does not make sense for this particular computation, but in many cases you would like the parser to just skip the faulty line and continue with the next. For example, if you use the parser to implement a compiler, you want it to continue and possibly discover more errors. Such behavior can be achieved by defining a faulty line as an additional alternative for `Line`, to be tried after `Store` and `Print` failed. With this addition, the first lines of your grammar would look like this (note the use of the shorthand operator `++`):

```
Input   = [\n\r]* Line++ EOF ;
Line    = Space (Store / Print) ~{badLine}
        / BadLine ;
Store   = Name Space Equ Sum EOL {store} ;    // store result
Print   = Sum EOL {print} ;                   // print result
BadLine = _++ EOL ;                           // skip bad line
```

The `BadLine` is defined to be any sequence of characters up to the nearest end of line or end of input. After a failure of `(Store / Print)`, the parser backtracks and consumes all of `BadLine`.

The name in brackets preceded by a tilde: `~{badLine}` identifies a semantic action to be invoked in case of failure. Thus, `badLine` is invoked when the first alternative of `Line` fails. The method has an access to a Phrase object for `Line`, appearing as the left-hand side of the failing rule `Line = (Store / Print)`. There is no right-hand side, but `lhs()` contains error information. The function of `badLine` is to print out this information and erase it. This is done by calling helper methods `errMsg()` and `errClear()`:

```
//--------------------------------------------------------------------
//  Line = Store / Print (failed)
//--------------------------------------------------------------------
void badLine()
  {
    System.out.println(lhs().errMsg());
    lhs().errClear();
 }
```

(The line will not be rescanned, so it is safe to print the message.)

The use of `++` in `Input` prevents a repeated call to `BadLine` after an erroneous last line. Because `BadLine` is called in an iteration, it must consume at least one character. It is therefore defined using `++` rather than `*+`. You may verify that it will never be called at `EOL`.

You can find the modified grammar and semantics in `example10`, together with the source for `FileCalc`. Generate the new parser in the usual way and compile everything; then damage the input file in several places, and run the parser on it using `FileCalc` to see the diagnostics.

18 Miscellaneous features

This section describes some features not mentioned in the tutorial.

18.1 Escaped characters

None of the characters you write in String Literal, Character Class, and Character Range is allowed to be newline or carriage return. The same applies to diagnostic names in pointed brackets. It is also not advisable to use there any white-space characters other than blank. Other characters, namely `"`, `]`, `-`, `>`, cannot be used because they will be mistaken for delimiters. Still other characters may be not available at your keyboard. You specify all such characters using the conventions of "escape".

Any character other than `n`, `r`, `t`, or `u`, preceded by a backslash `\` represents itself as part of the string s in String Literal and Character Class, or as c_1, c_2 in Character Range, but is not recognized as a delimiter. Thus, for example, `"\""` is a String Literal of one double quote `"`, and `[[\]]` is a Character Class consisting of the brackets `[` and `]`.

The special combinations `\n`, `\r`, and `\t` represent the newline, carriage return, and tab characters, respectively. The special combination `\u` must be followed by four hexadecimal digits representing a Unicode character. Thus, for example, `\u000B` stands for the vertical tab character.

18.2 Character encoding

Both your grammar file and the file processed by your parser consist of bytes that may be interpreted in different ways depending on the used *character encoding scheme*. For example, the byte with value 248 (hex `f8`) means the Norwegian character `ø` in the encoding ISO-8859-1, or the Czech character `ř` in the encoding ISO-8859-2; the command prompt window of Windows XP displays it as a degree sign °. The bytes with values in the range 32 through 126, known as the ASCII character set, have a fixed interpretation that does not depend of encoding.

Both the `Generate` tool and the resulting parser use the default character encoding set up for the Java Virtual Machine executing them. Your grammar uses, in principle, only the ASCII characters, but you can have other characters in values of terminals, in diagnostic names, or in comments. These chracters are interpreted according to the default encoding. When they appear as literals in the generated Java file, they are converted to Unicode escapes that are composed of ASCII characters. It means that the compiled parser does not depend on the encoding used at the compile time.

The input file processed by the parser is also interpreted according to the default encoding. If you need, you may enforce a specific encoding for input files by modifying the wrapper `SourceFile`.

18.3 Reserved names

The names you give to expressions and syntactic actions are used as method and object names in the generated parser. Therefore they must be distinct from Java reserved words. They may also conflict with names used in the base classes, but no checks are made in the present version, and **you have, unfortunately, to watch yourself for possible conflicts!**

18.4 Boolean actions

You may specify semantic actions that return a `boolean` value in addition to inserting semantic value in the left-hand side `Phrase`. The action returning `false` causes the invoking procedure to fail. You specify such action by placing an ampersand in front of the action name. For example, the action `identifier()` invoked by `Identifier` below may return `false` if the recognized text is one of reserved words, thus forcing `Identifier` to fail on reserved words:

```
Identifier = Letter (Letter / Digit)*  {& identifier} ;
```

18.5 Omitted action names

To avoid clobbering your grammar with names of semantic actions, you may only indicate the presence of an action with an empty pair of brackets `{}`. The semantic action thus specified has the same name as the rule. The actions for multiple alternatives of a choice expression will have numbers attached to them. For example, you can write:

```
Product = Factor (MultOp Factor)*  {} ;
Factor  = Digits? "." Digits Space {}
        / Digits Space             {}
        / Lparen Sum Rparen        {}
        / Name Space               {} ;
```

The semantic actions defined here are `Product()` and `Factor_0()` through `Factor_3()`. These names will appear in the skeleton generated by `-s` option.

You specify a Boolean action by an ampersand in brackets: `{&}`.

You specify an error action by brackets preceded with a tilde: `~{}`.
The name of the action thus specified has suffix `"_fail"`.

18.6 Own diagnostic output

The error messages described in Section 14 are written to `System.out`. If this does not suit you, you may yourself take care of the message as shown in Section 17. You define a semantic action to be called on failure of the top procedure. In that action, you obtain the error message using `lhs().errMsg()`, and handle it in the way you need. You should then issue `lhs().errClear()` to prevent the message from being written to `System.out`.

The messages returned by `errMsg()` may contain unprintable characters appearing in your String Literals, Character Classes, and Character Ranges. In the `String` returned by `errMsg()`, characters outside the range `\u0020` through `\u00ff` are replaced by Java-like escapes such as `\n` for a newline character, or `\u03A9` for the Greek letter Ω. Characters in the range `\u007f` through `\u00ff` are not replaced; they may be displayed differently depending on the encoding used to view the message.

(The above applies to all messages from the parser.)

18.7 Trace

The base class `mouse.runtime.SemanticsBase` contains a `String` variable `trace`, normally initialized to empty string. Using the method `setTrace()` of your parser, or the option `-T` on *Mouse* tools `TryParser` and `TestParser`, you can assign a value to this variable in order to trigger special actions such as tracing.

Note that the same value is used to trigger traces in the test version of the parser; therefore, you should not use strings containing letters `e`, `i`, or `r` if you are going to use this feature together with the test version.

18.8 Package name

In a real application, your parser and semantics classes will belong to some named package. You specify the name of that package using option `-p` on `mouse.Generate`.

18.9 Initialization

The base class `mouse.runtime.SemanticsBase` contains an empty method `void Init()` that is invoked every time you start your parser by invoking its `parse` method. By overriding this method in your semantics class, you can provide your own initialization of that class.

18.10 Specifying output directory

By default, the `Generate` tool places the resulting parser (and semantics skeleton if requested) in the current work directory. This default has been used in all the examples. You can specify any output directory of your choice using the `-D` option on `mouse.Generate`. The name of this directory need not correspond to the specified package name.

18.11 Checking the grammar

As mentioned in Section 2, *Mouse* checks if the grammar is well-formed, thus ensuring that the generated parser will always stop. As an example, the following grammar is not well-formed:

```
A = (!"a")* / "b"? A ;
```

As `"b"?` may succeed without consuming any input, there is a possibility of infinite descent for `A`. As `!"a"` may succeed without consuming input, there is a possibility infinite iteration in `(!"a")*`. The `Generate` tool produces these messages:

```
Warning: the grammar not well-formed.
- !"a" in (!"a")* may consume empty string.
- A is left-recursive via "b"? A.
```

You may use the *Mouse* tool `TestPEG` to check the grammar without generating parser. By specifying option `-D` to the tool, you display the grammar together with attributes that were used to compute well-formedness according to [7]. (The rules for computing these attributes can also be found as Figs. 5 and 6 in [11].)

```
java mouse.TestPEG -G badGrammar.txt -D
Warning: the grammar not well-formed.
- !"a" in (!"a")* may consume empty string.
- A is left-recursive via "b"? A.

1 rules
  A = (!"a")* / "b"? A ;   //  0 !WF

4 subexpressions
  (!"a")*   //  0 !WF
  "b"?   //  01
  !"a"   //  0f
  "b"? A   //  01 !WF

2 terminals
  "a"   //  1f
  "b"   //  1f
```

The symbols appearing as comments on the right have these meanings:

`0` = may succeed without consuming input;
`1` = may succeed after consuming some input;
`f` = may fail;
`!WF` = not well-formed.

19 Deploying the parser

To include the generated parser in your application, you need to include your parser and semantics class, in the package of your choice (specified by the `-p` option on `mouse.Generate`).

You also need the standard *Mouse* package `mouse.runtime`. It is included in the `JAR` file `Mouse-1.5.1.jar`, and is also provided as a separate `JAR` file `Mouse-1.5.1.runtime.jar`. It is sufficient to have any of them accessible via `CLASSPATH`. You may also extract the directory `mouse/runtime` from any of these `JAR` files and make it accessible via `CLASSPATH`.

Alternatively, you can integrate the runtime support into your package. You can use for this purpose the *Mouse* tool `MakeRuntime`. For example, the command:

```
mouse.MakeRuntime -r my.runtime
```

produces in the current work directory the source code of all runtime support classes, with declaration `"package my.runtime;"`. By specifying the option `-r my.runtime` on the invocation of `Generate`, your generated parser will be an extension of `my.runtime.ParserBase` rather than `mouse.runtime.ParserBase`. The generated skeleton of your semantics class will be an extension of `my.runtime.SemanticsBase`. (If you create the class on your own, you have to declare it so.)

The runtime support consists of nine classes:

- `ParserBase`.
- `SemanticsBase`.
- `ParserMemo` – extension of `ParserBase`.
- `ParserTest` – extension of `ParserTest`.
- `CurrentRule` – interface for accessing parser stack.
- `Phrase` – interface for accessing `Phrase` objects.
- `Source` – interface of input wrappers.
- `SourceFile` – wrapper for input from a file.
- `SourceString` – wrapper for input from a `String`.

Of these, you need `ParserMemo` only if you generate the memoizing (`-M`) or the instrumented (`-T`) version of the parser. You need `ParserTest` only for the instrumented version. Of `SourceFile` and `SourceString` you need only the one that you use.

`SourceFile` assumes that the input file uses default character encoding. You can change it by modifying `SourceFile.java` in the place indicated by a comment.

A Appendix: The grammar of *Mouse* PEG

```
Grammar   = Space (Rule/Skip)*+ EOT ;
Rule      = Name EQUAL RuleRhs DiagName? SEMI ;
Skip      = SEMI
          / _++ (SEMI/EOT) ;
RuleRhs   = Sequence Actions (SLASH Sequence Actions)* ;
Choice    = Sequence (SLASH Sequence)* ;
Sequence  = Prefixed+ ;
Prefixed  = PREFIX? Suffixed ;
Suffixed  = Primary (UNTIL Primary / SUFFIX)? ;
Primary   = Name
          / LPAREN Choice RPAREN
          / ANY
          / StringLit
          / Range
          / CharClass ;
Actions   = OnSucc OnFail ;
OnSucc    = (LWING AND? Name? RWING)? ;
OnFail    = (TILDA LWING Name? RWING)? ;
Name      = Letter (Letter / Digit)* Space ;
DiagName  = "<" Char++ ">" Space ;
StringLit = ["] Char++ ["] Space ;
CharClass = ("[" / "^[") Char++ "]" Space ;
Range     = "[" Char "-" Char "]" Space ;
Char      = Escape
          / ^[\r\n\\] ;
Escape    = "\\u" HexDigit HexDigit HexDigit HexDigit
          / "\\t"
          / "\\n"
          / "\\r"
          / !"\\u""\\"_ ;
Letter   = [a-z] / [A-Z] ;
Digit    = [0-9] ;
HexDigit = [0-9] / [a-f] / [A-F] ;
PREFIX   = [&!]  Space ;
SUFFIX   = [?*+] Space ;
UNTIL    = ("*+" / "++") Space ;
EQUAL   = "=" Space ;
SEMI    = ";" Space ;
SLASH   = "/" Space ;
AND     = "&" Space ;
LPAREN  = "(" Space ;
RPAREN  = ")" Space ;
LWING   = "{" Space ;
RWING   = "}" Space ;
TILDA   = "~" Space ;
ANY     = "_" Space ;
Space   = ([ \r\n\t] / Comment)* ;
Comment = "//" _*+ EOL ;
EOL     = [\r]? [\n] / !_  ;
EOT     = !_  ;
```

B Appendix: Helper methods

These methods provide access to the environment seen by a semantic action.

The following four methods are inherited from `mouse.runtime.Semantics`.
They call back the parser to access the relevant `Phrase` objects.

`Phrase` **lhs**`()`
Returns the left-hand side object.

`int` **rhsSize**`()`
Returns the number of right-hand side objects.

`Phrase` **rhs**`(int i)`
Returns the i-th element on the right-hand side, $0 \leq i <$ `rhsSize()`.

`String` **rhsText**`(int i,int j)`
Returns as one `String` the text represented by the right-hand side objects i through $j-1$, where $0 \leq i < j \leq$ `rhsSize()`.

The following eleven methods are defined by the interface `mouse.runtime.Phrase`.

`void` **put**`(Object v)`
Inserts v as the semantic value of this `Phrase`.

`Object` **get**`()`
Returns the semantic value of this `Phrase`.

`String` **text**`()`
Returns the text represented by this `Phrase`.

`char` **charAt**`(int i)`
Returns the i-th character of the text represented by this `Phrase`.

`String` **rule**`()`
Returns name of the rule that created this `Phrase`.

`boolean` **isA**`(String name)`
Returns `true` if this `Phrase` was created by rule *name*.

`boolean` **isTerm**`()`
Returns `true` if this `Phrase` was created by a terminal.

`boolean` **isEmpty**`()`
Returns `true` if the text represented by this `Phrase` is empty.

`String` **errMsg**`()`
Returns the error message contained in this `Phrase`.

`void` **errClear**`()`
Erases the error message contained in this `Phrase`.

`String` **where**`(int i)`
Returns a printable string describing where to find the i-th character of text represented by this `Phrase` in the text being parsed.

C Appendix: Your parser class

These are the methods you can apply to your generated parser.
"`Parser`" and "`Semantics`" are names of your parser and semantics classes, respectively.

Parser`()`
Parser constructor. Instantiates your parser and semantics, connects semantics object to the parser, and returns the resulting parser object.

`boolean` **parse**`(Source src)`
Parses input wrapped into a `Source` object *src*.
Returns `true` if the parse was successful, or `false` otherwise.

`Semantics` **semantics**`()`
Returns the semantics object associated with the parser.

`void` **setTrace**`(String s)`
Assigns *s* to the `trace` field in semantics object.

`void` **setMemo**`(int n)`
Sets the amount of memoization to *n*.
Can only be applied to a parser generated with option `-M` or `-T` (see `mouse.Generate` tool).
For a parser generated with `-M` *n* can have values from 1 to 9.
For a parser generated with `-T` *n* can have value from 0 to 9.

D Appendix: *Mouse* tools

mouse.Generate *arguments*

Generates parser.
The *arguments* are specified as options according to POSIX syntax:

`-G` *filename*
Identifies the file containing your grammar. Mandatory.
The *filename* need not be a complete path, just enough to identify the file in current environment. It should include file extension, if any.

`-D` *directory*
Identifies target directory to receive the generated file(s).
Optional. If omitted, files are generated in current work directory.
The *directory* need not be a complete path, just enough to identify the directory in current environment. The directory must exist.

`-P` *parser*
Specifies name of the parser to be generated. Mandatory.
Must be an unqualified class name. The tool generates a file named "*parser*`.java`" in the target directory, The file contains definition of Java class *parser*. If target directory already contains a file "*parser*`.java`", the file is replaced without a warning,

`-S` *semantics*
Indicates that semantic actions are methods in the Java class *semantics*.
Mandatory if your grammar specifies semantic actions. Must be an unqualified class name.

`-p` *package*
Generate parser as member of package *package*.
The semantics class, if specified, is assumed to belong to the same package.
Optional. If not specified, both classes belong to unnamed package.
The specified package need not correspond to the target directory.

`-r` *runtime-package*
Generate parser that will use runtime support package *runtime-package*.
Optional. If not specified, `mouse.runtime` is assumed.

`-s` Generate skeleton of semantics class. Optional.
The skeleton is generated as file "*semantics*`.java`" in the target directory, where *semantics* is the name specified by `-S` option. The option is ignored if `-S` is not specified. If target directory already contains a file "*semantics*`.java`", the tool is not executed, to prevent accidental destruction of your semantics class.

`-M` Generate memoizing version of the parser.

`-T` Generate instrumented test version of the parser.

(Options `-M` and `-T` are mutually exclusive.)

mouse.TestPEG *arguments*

Checks your grammar without generating parser.
The *arguments* are specified as options according to POSIX syntax:

`-G` *filename*
Identifies the file that contains your grammar. Mandatory.
The *filename* should include any extension.
Need not be a complete path, just enough to identify the file in your environment.
`-D` Display the grammar. Optional.
Shows the rules and subexpressions together with their attributes according to Ford.
(`0`=may consume empty string, `1`=may consume nonempty string, `f`= may fail, `!WF`=not well-formed.)
`-C` Display the grammar in compact form: without duplicate subexpressions. Optional.
`-R` Display only the rules. Optional.

(Obviously, `-C`, `-D`, and `-R` are mutually exclusive.)

mouse.TryParser *arguments*

Runs the generated parser.
The *arguments* are specified as options according to POSIX syntax:

`-P` *parser*
Identifies the parser. Mandatory.
parser is the class name, fully qualified with package name, if applicable. The class must reside in a directory corresponding to the package.
`-f` *filename*
Apply the parser to file *filename*. Optional.
The *filename* should include any extension.
Need not be a complete path, just enough to identify the file in your environment.
`-F` *list*
Apply the parser separately to each file in a list of files. Optional.
The *list* identifies a text file containing one fully qualified file name per line. The *list* itself need not be a complete path, just enough to identify the file in your environment.
`-m` *n*
Amount of memoization. Optional. Applicable only to a parser generated with option `-M`.
n is a digit from 1 through 9 specifying the number of results to be cached. Default is `-m1`.
`-T` *string*
Tracing switches. Optional.
The *string* is assigned to the `trace` field in your semantics object, where you can use it to activate any trace you have programmed there.

If you do not specify `-f` or `-F`, the parser is executed interactively, prompting for input by printing "`>`". It is invoked separately for each input line after you press "Enter". You terminate the session by pressing "Enter" directly at the prompt.

mouse.TestParser *arguments*

Runs the instrumented parser (generated with option `-T`), and prints information about its operation, as explained in Sections 9, 10, and 15.

`-P` *parser*
Identifies the parser. Mandatory.
parser is the class name, fully qualified with package name, if applicable. The class must reside in a directory corresponding to the package.

`-f` *filename*
Apply the parser to file *filename*. Optional.
The *filename* should include any extension.
Need not be a complete path, just enough to identify the file in your environment.

`-F` *list*
Apply the parser separately to each file in a list of files. Optional.
The *list* identifies a text file containing one fully qualified file name per line. The *list* itself need not be a complete path, just enough to identify the file in your environment.

`-m` *n*
Amount of memoization. Optional.
n is a digit from 0 through 9 specifying the number of results to be retained. Default is `-m0`.

`-T` *string*
Tracing switches. Optional,
The *string* is assigned to the `trace` field in your semantics object, where you can use it to trigger any trace you have programmed there. In addition, presence of certain letters in *string* activates traces in the parser:
`r` – trace execution of parsing procedures for rules.
`i` – trace execution of parsing procedures for inner expressions.
`e` – trace error information.

`-d` Show detailed statistics for procedures involved in backtracking - rescan - reuse. Optional.

`-D` Show detailed statistics for all invoked procedures. Optional.

(Options `-d` and `-D` are mutually exclusive.)

If you do not specify `-f` or `-F`, the parser is executed interactively, prompting for input by printing "`>`". It is invoked separately for each input line after you press "Enter". You terminate the session by pressing "Enter" directly at the prompt.

mouse.MakeRuntime *arguments*

Re-package *Mouse* runtime support: write out Java source files for runtime-support classes using specifed package name instead of `mouse.runtime`.
The *arguments* are specified as options according to POSIX syntax:

`-r` *runtime-package*
Generate files with package name *runtime-package*. Mandatory.

`-D` *directory*
Identifies target directory to receive the files.
Optional. If omitted, files are written to the current work directory.
The *directory* need not be a complete path, just enough to identify the directory in current environment. The directory must exist.
The directory name need not correspond to the package name.

References

[1] Aho, A. V., and Ullman, J. D. *The Theory of Parsing, Translation and Compiling, Vol. I, Parsing*. Prentice Hall, 1972.

[2] Birman, A. *The TMG Recognition Schema*. PhD thesis, Princeton University, February 1970.

[3] Birman, A., and Ullman, J. D. Parsing algorithms with backtrack. *Information and Control 23* (1973), 1–34.

[4] Brooker, P., and Morris, D. Some proposals for the realization of a certain assembly program. *The Computer Journal 3*, 4 (1961), 220–231.

[5] Ford, B. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002.

[6] Ford, B. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming* (October 2002).

[7] Ford, B. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy, 14–16 January 2004), N. D. Jones and X. Leroy, Eds., ACM, pp. 111–122.

[8] Grimm, R. Rats! – an easily extensible parser generator. `http://www.cs.nyu.edu/rgrimm/xtc/rats.html`.

[9] Grimm, R. Practical packrat parsing. Tech. Rep. TR2004-854, Dept. of Computer Science, New York University, March 2004.

[10] Hopgood, F. R. A. *Compiling Techniques*. MacDonalds, 1969.

[11] Koprowski, A., and Binsztok, H. TRX: A formally verified parser interpterer. In *ESOP 2010: 19th European Symposium on Programming* (Paphos, Cyprus, 22–26 Mar. 2010), A. D. Gordon, Ed., no. 6012 in Lecture Notes in Comp. Sci., Springer-Verlag, pp. 345–365.

[12] Lucas, P. The structure of formula-translators. *ALGOL Bulletin Supplement 16* (September 1961), 1–27.

[13] McClure, R. M. TMG – a syntax directed compiler. In *Proceedings of the 20th ACM National Conference* (24–26 August 1965), L. Winner, Ed., ACM, pp. 262–274.

[14] Michie, D. Memo functions and machine learning. *Nature 218* (April 1968), 19–22.

[15] Redziejowski, R. R. Parsing Expression Grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae 79*, 3–4 (2007), 513–524.

[16] Redziejowski, R. R. Some aspects of Parsing Expression Grammar. *Fundamenta Informaticae 85*, 1–4 (2008), 441–454.

[17] Redziejowski, R. R. Applying classical concepts to Parsing Expression Grammar. *Fundamenta Informaticae 93*, 1–3 (2009), 325–336.

[18] Rosen, S. A compiler-building system developed by Brooker and Morris. *Commun. ACM 7*, 7 (July 1964), 403–414.

[19] Schmitz, S. Modular syntax demands verification. Tech. Rep. I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice - Sophia Antipolis, Oct. 2006.