Mouse: From Parsing Expressions to a Practical Parser

Roman R. Redziejowski

Appeared in *Proceedings of the CS&P'2009 Workshop* vol. 2,
Ed. L.Czaja and M.Szczuka, Warsaw University (2009) 514–525

Abstract. 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 directly define the structures that usually require a separate "lexer" or "scanner". This paper outlines a tool, named *Mouse*, to transcribe PEG into an executable parser written in Java. 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. The name "*Mouse*" sets this tool apart from another parser generator named "*Rats!*" that produces a storage-hungry "packrat parser".

1 Introduction

Recursive-descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. It was suggested as early as in 1961 by Lucas [6]. The great advantage of a recursive-descent parser is its simplicity and clear relationship to the grammar. For smaller grammars, the parser can be easily produced and maintained by hand. This is contrary to bottom-up parsers, normally driven by large tables that have no obvious relationship to the grammar; these tables *must* be mechanically generated.

The problem with constructing recursive-descent parsers from a classical context-free grammar is that the grammar must have the so-called $LL(1)$ property. 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. However, full backtracking may require exponential time. A reasonable compromise is limited backtracking: never try another alternative after one alternative already succeeded on a portion of input.

Recently, Ford [1–3] introduced a language for writing recursive-descent parsers with limited backtracking. It is called Parsing Expression Grammar (PEG) and has the form of a grammar that can be easily transcribed into a set of recursive procedures.

In addition to backtracking, PEG can directly define structures that normally require a separate "lexer" or "scanner". Together with lifting of the $LL(1)$ restriction, this gives a very convenient tool when we need an ad-hoc parser for some application.

Theoretically, even the limited backtracking may require a lot of time. In [1, 2], PEG was introduced together with a technique called *packrat parsing*. Packrat parsing handles backtracking by extensive *memoization*: storing all results of parsing procedures.

It guarantees linear parsing time at a huge memory cost. There exists a complete parser generator named *Rats!* [4, 5] 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, experiments reported in [7, 8] demonstrated a moderate backtracking activity in PEG parsers for programming languages Java 1.5 and C.

This paper presents a tool, named *Mouse*, that translates PEG into a set of recursive procedures. The result is a straightforward code that closely reflects the grammar. It is not a packrat parser. The run-time storage requirements are limited to the usual parser stack.

Both *Mouse* and the resulting parser are written in Java.

An integral feature of *Mouse* is the mechanism for specifying semantics, also in Java. Unlike many parser generators, *Mouse* does not provide automatic generation of syntax trees. However, one can always do it by specifying suitable semantics, and create trees that best suit the particular purpose.

Complete documentation can be found in the user's manual [10]. All materials can be downloaded from the site `http://www.romanredz.se/freesoft.htm`.

2 Parsing Expression Grammar

Parsing Expression Grammar is a collection of "parsing expressions".

Each parsing expression specifies a procedure to recognize certain text patterns. The procedure tries to recognize a portion of input text. If it succeeds, it "consumes" the recognized portion and indicates "success". Otherwise it indicates "failure" and does not consume anything.

`"`s`"`	Recognize string s.
`[`s`]`	Recognize any character appearing in string s.
`[`c_1`-`c_2`]`	Recognize any character from the range c_1–c_2.
`_`	Recognize any character (fails only at the end of input).
e`?`	Call e and return success whether it succeeded or not.
e`*`	Call e repeatedly as long as it succeeds. Always return success.
e`+`	Call e repeatedly as long as it succeeds. Return success if e succeeded at least once.
`&`e	Look-ahead: call e and backtrack. Return success if e succeeded.
`!`e	Look-ahead: call e and backtrack. Return success if e failed.
$e_1 \ldots e_n$	Sequence: Call $e_1, \ldots, e_n$, in this order. Backtrack if any of them fails.
e_1 `/` $\ldots$ `/` e_n	Choice: Call $e_1, \ldots, e_n$, in this order, until one of them succeeds.

Figure 1. Parsing Expressions.

A parsing expression can have one of the eleven forms listed in Figure 1. The first four directly test the input. The remaining seven do their job by calling other expressions. These expressions, denoted by $e, e_1, \ldots, e_n$ in the Figure, can be specified either by name or explicitly, enclosed in parentheses if needed. For a precise definition, the reader is referred to [3, 7–9].

The grammar is supplied to *Mouse* as one or more "rules" of the form:

name `=` *expr* `;`

where *expr* is a parsing expression, and *name* is a name given to it. 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 appearing first defines the "top procedure", invoked at the start of the parser.

A specific grammar may look like this:

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

It defines a parser consisting of six named procedures. One can easily see that the parser accepts one or more integers separated by plus and terminated by end of input. It allows white space around the items; notice that this is a feature usually delegated to a separate "lexer".

3 Generated parser

The parser generated by *Mouse* is a Java class with name given by the user, for example `"myParser"`.

The procedures that implement parsing expressions are methods in that class. They have names of the expressions they implement. Procedures that implement inner expressions have generated names.

Figure 2 shows fragments of the parser generated from the example given above. Notice the procedure `Sum_0` generated for the inner expression `(Plus Number)`.

The generated parser is a subclass of `mouse.runtime.ParserBase` provided by *Mouse*. The class `ParserBase` contains standard services such as parser stack, access to input, initialization, error handling. The methods `begin`, `accept`, `reject`, `acceptInner`, `rejectInner`, and `next` used in the parsing procedures are inherited from that class.

The method `begin` pushes a new entry on the parser stack. Methods `accept` and `acceptInner` handle successful termination of a named, respectively inner, expression and return `true`. Methods `reject` and `rejectInner` handle unsuccessful termination and return `false`. They perform backtracking when applicable.

The method `next` is an example of built-in procedures that directly access the input. The particular call `next('+')` checks if the next character is `"+"` and consumes it if so.

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

boolean Sum_0()            // Plus Number
  {
    begin("");
    if (!Plus()) return rejectInner();
    if (!Number()) return rejectInner();
    return acceptInner();
  }

boolean Plus()             // Plus = "+" Space
  {
    begin("Plus");
    if (!next('+')) return reject();
    if (!Space()) return reject();
    return accept();
  }
```

Figure 2. Examples of parsing procedures.

4 Specifying semantics

The semantics is defined by providing "semantic actions" that will be invoked from parsing procedures. Semantic actions are methods in a separate class written by the user. This class must be a subclass of `mouse.runtime.SemanticsBase` provided by *Mouse*. It can have any name chosen by the user, for example `"mySemantics"`. The class `SemanticsBase` contains "helper methods" that give access to results of the parsing. Figure 3 shows the class structure of a parser with semantics.

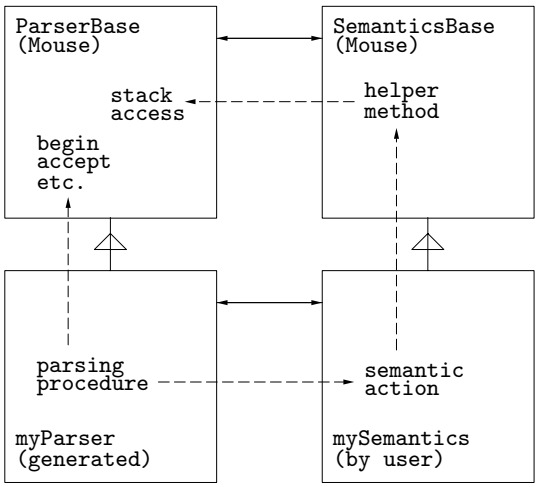

Figure 3. Parser with semantics.

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

The calls to semantic actions are indicated in the grammar by adding the action's name in braces at the end of a rule. Suppose we want to provide semantic actions for `Sum` and `Number`, named, respectively, `sum` and `number`. This is indicated in the grammar as follows:

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

It results in *Mouse* generating this code for the two parsing procedures:

```
boolean Sum()              // Sum = Number (Plus Number)*
  {
    begin("Sum");
    if (!Number()) return reject();
    while (Sum_0());
    sem.sum();
    return accept();
  }

boolean Number()           // Number = Digits Space
  {
    begin("Number");
    if (!Digits()) return reject();
    if (!Space()) return reject();
    sem.number();
    return accept();
  }
```

where `sem` is a reference to `mySemantics`, available in `myParser`. As one can see, each procedure calls its semantic action as the last thing before a successful return.

An outline of the corresponding class `mySemantics` is shown in Figure 4. We are now going to fill in the details indicated by dots in the Figure. We do it under assumption that the purpose is to compute the sum of consumed integers and print it out. Quite naturally, the job of `number()` is to convert the integers into the form that can be used in the computation, and the job of `sum()` is to do the actual computation.

```
class mySemantics extends SemanticsBase
{
  //-------------------------------------------------------
  //  Number = Digits Space
  //-------------------------------------------------------
  void number()
    { ... }

  //-------------------------------------------------------
  //  Sum = Number (Plus Number)*
  //-------------------------------------------------------
  void sum()
    { ... }
}
```

Figure 4. Outline of semantics class.

To do their job, semantic actions must have access to the result of parsing. Behind the scenes, `ParserBase` creates for each call to parsing procedure a `Phrase` object – an instance of class `Phrase`. This class is an internal class of `ParserBase`. Each `Phrase` object represents the portion of input consumed by the specific procedure call – the consumed "phrase". It contains the starting and ending position of the phrase in the input text. It also contains the name of the procedure as a `String`, and a field of type `Object` where the user can place a "semantic value".

`ParserBase` maintains a stack of `Phrase` objects representing the currently active procedures. To describe the details, we are going to illustrate what happens when the parser processes the input `"4711␣+␣17"` (where ␣ stands for blank space).

Figure 5 shows the stack at the moment when the parser processed the initial `"4711␣"`. Each box represents one `Phrase` object. The string at the bottom of the box is the consumed text. It is *not* represented by a `String` inside the object, only by the starting and ending positions in the input.

The stack is the chain of `Phrase`s starting with `Input` at the bottom. `Phrase` objects on the right represent partial results.

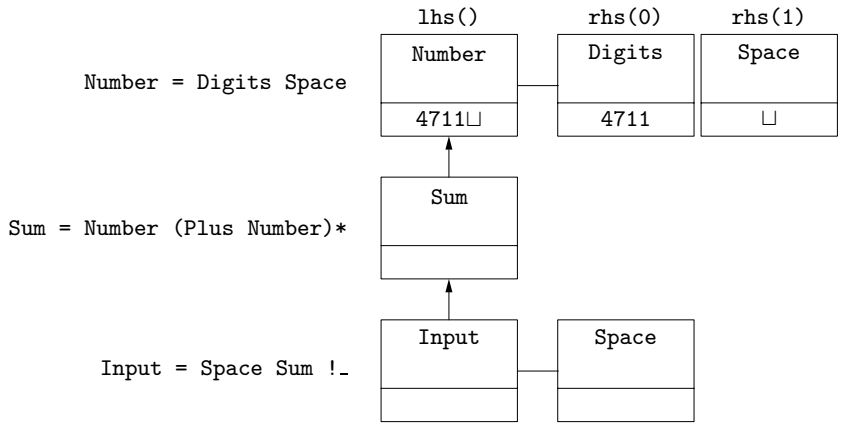

Figure 5. The stack after processing `"4711␣"`, before call to `number()`.

The processing started by calling `Input` that consumed empty string by a call to `Space()`. The `Phrase` object on the right is the result of that call. `Input()` called then `Sum()` that immediately called `Number()`. This latter consumed `"4711␣"` by successful calls to `Digits()` and `Space()`, but did not return yet to its caller. The results of `Digits()` and `Space()` appear on the right. This is the moment where `Number()` calls its semantic action `number()`.

Through helper methods, `number()` can access the three `Phrase`s on top of Figure 5. They are obtained, respectively, as `lhs()`, `rhs(0)`, and `rhs(1)`, which stand for "left-hand side", "right-hand side element 0", and "right-hand side element 1".

As indicated before, `number()` is expected to convert the text `"4711"` consumed by `Digits()` to an integer 4711. As we shall see in a short while, the `Phrase` object for `Number` that appears as the "left-hand side" in Figure 5 will eventually become a part of the "right-hand side" seen by the semantic action `sum()`. We can pass the integer 4711 to it as the semantic value of that object.

The text consumed by a `Phrase` is obtained by calling the method `text()` of that `Phrase`. The text `"4711"` is thus obtained as `rhs(0).text()`. A semantic value is inserted into a `Phrase` by means of the method `put(...)`. The required action is thus performed by a single statement:

```
lhs().put(Integer.valueOf(rhs(0).text()));
```

After calling its semantic action, `Number()` terminates by calling `accept`. This removes the `Number`'s `Phrase` from the stack and attaches it as a partial result of `Sum`. The `Digits` and `Space` objects are discarded. The resulting stack is shown in Figure 6. Notice the semantic value 4711 in the `Number`'s `Phrase`.

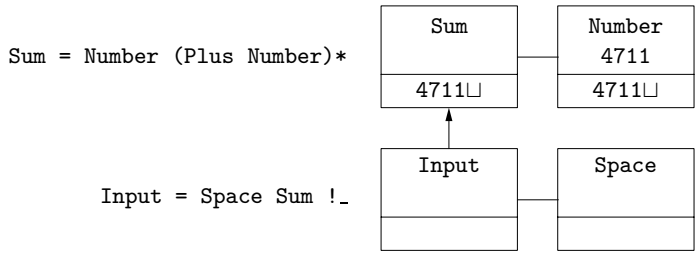

Figure 6. The stack after processing `"4711␣"` and return from `Number()`.

After receiving control back from `Number()`, `Sum()` enters its `while` statement that will repeatedly call `Sum_0()`. Figure 7 shows the stack when the first call to `Sum_0()` already consumed `"+␣17"` by successful calls to `Plus()` and `Number()`. Notice the semantic value 17 that was placed in the new `Number` object by semantic action `number()`.

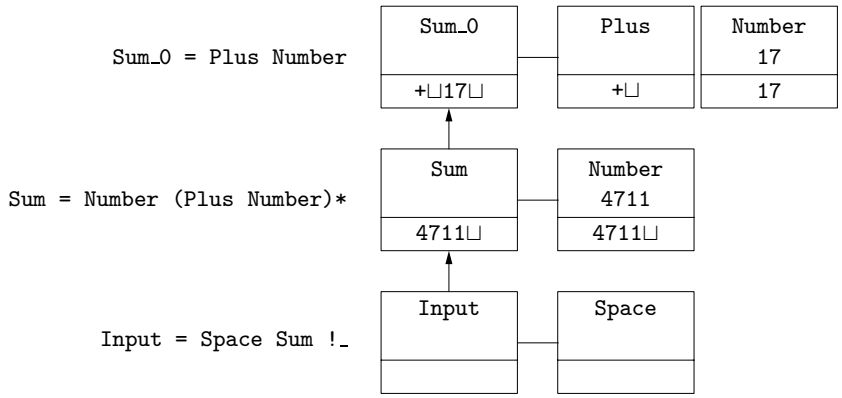

Figure 7. The stack after processing `"4711␣+␣17"`, before return from `Sum_0()`.

After receiving control back from `Number()`, `Sum_0()` exits by calling `acceptInner` that handles successful return from procedures for inner expressions. Inner expressions return their partial results to the caller, so the results of `Plus()` and `Number()` are

attached to `Sum` and the object for `Sum_0()` is discarded. The resulting stack is shown in Figure 8.

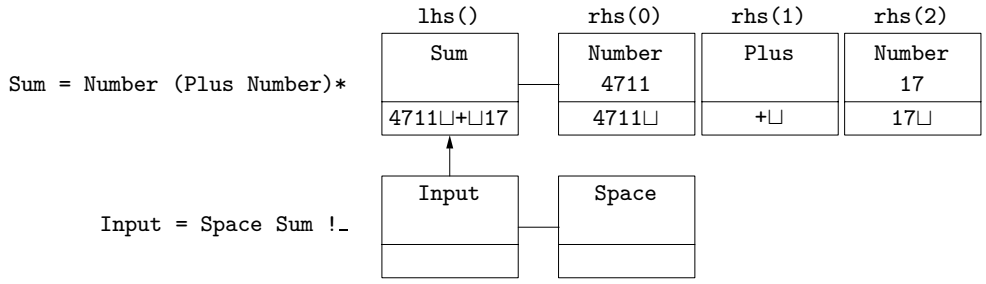

Figure 8. The stack after processing `"4711␣+␣17"`, before call to `sum()`.

The next call to `Sum_0()` encounters the end of text and fails, thus terminating the `while` statement, and `Sum()` proceeds to call its semantic action `sum()`. This latter has access to the four objects on top of Figure 8 and can compute the sum of the two semantic values appearing in `rhs(0)` and `rhs(2)`.

We must remember, however, that in the general case there will be k `(Plus Number)` pairs consumed by $k \geq 0$ invocations of `Sum_0`. That means we shall in general have $n = 2k + 1$ right-hand side elements numbered like this:

```
Number Plus   Number  ... Plus    Number
rhs(0) rhs(1) rhs(2)     rhs(n-2) rhs(n-1)
```

The number n is obtained by calling the helper method `rhsSize()`. The semantic value of a `Phrase` is obtained by calling the method `get()` that returns an `Object`. The required computation is thus done as follows:

```
int s = 0;
for (int i=0;i<rhsSize();i+=2)
  s += (Integer)rhs(i).get();
System.out.println(sum);
```

The complete class `mySemantics` that computes the sums is shown below.

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

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

5 Generating the parser

To construct a parser, you use the *Mouse* utility `Generate` that is invoked from command line like this:

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

In this invocation, "`myGrammar.txt`" names the file containing the grammar, "`myParser`" is name to be given to the generated parser, and "`mySemantics`" is name of the semantics class. The utility creates a file `myParser.java` containing the parser.

Optionally, the utility may also generate a skeleton of the semantics class in a file `mySemantics.java`, looking like Figure 4 with empty method bodies.

6 Backtracking

To illustrate backtracking, we add an alternative number format to our toy grammar:

```
Input  = Space Sum !_ ;
Sum    = Number (Plus Number)* ;
Number = Digits? "." Digits Space  // fraction
       / Digits Space ;            // integer
Plus   = "+" Space ;
Digits = [0-9]+ ;
Space  = " "* ;
```

The alternative format is a decimal fraction with or without digits before the decimal point. (Notice that this is another feature usually delegated to a separate "lexer".)

The definition of `Number` does not have the $LL(1)$ property: both alternatives may start with `Digits`. Encountering a sequence of digits followed by a blank or plus, `Number()` starts with the first alternative, and calls `Digits()` that consumes all the digits and constructs a `Phrase` to represent them. Not finding the decimal point, `Number()` discards the `Phrase`, backtracks to where it started, and tries the second alternative that again calls `Digits()` to repeat the same job. This is the price for circumventing the $LL(1)$ requirement.

You can watch the backtracking activity by generating an instrumented version of the parser. This is done by specifying an option to the `Generate` utility. The instrumented parser uses the same semantics class as the ordinary one. You execute the instrumented parser using another *Mouse* utility, `TestParser`. It produces this result for input "`123 + 4567`":

```
50 calls: 35 ok, 13 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
```

The first three lines tell that to process the input "`123 + 4567`", the parser executed 50 calls to parsing procedures, of which 35 succeeded, 13 failed, and two backtracked. 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.

The rest are detailed statistics for individual procedures that were involved in backtracking and rescanning. "`Number_0`" is the first alternative of `Number`. The heading "`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 the next section.

7 A mouse, not a pack rat

Optionally, *Mouse* can offer a small amount of memoization using the technique described in [7]. This is done by attaching to each parsing procedure a cache that can hold a small number (1-9) of the most recent `Phrase`s created by the procedure. As each `Phrase` contains a pointer to the input text, the procedure may find that it already has the result, and directly return the `Phrase`.

This function can be exercised by the instrumented parser by specifying, via an option, the size of the cache. Repeating the test from the preceding section with cache size 1 (that is, one most recent `Phrase` kept for each procedure) gives this result:

```
41 calls: 28 ok, 11 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            4     0     0     0     2     0     0
Number_0          0     0     2     0     0     7     4 After '123 + '
```

It shows 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 the instrumentation overhead. 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.

8 Error handling

8.1 Diagnostics

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.

This is implemented so that information about such "farthest failure(s)" is collected in the `Phrase` objects constructed by parsing procedures. Even a failing procedure returns a `Phrase` object to its caller, with information that it failed and why. The caller synthesizes new failure information from the received `Phrase`s and inserts it in its own `Phrase`.

A syntax error causes the parsing procedures to fail and backtrack to the start of input. This leaves the top procedure with its `Phrase` containing the error information; it is used to construct the final error message.

The following is an example of such message produced for input `"4711 # 17"` by the parser from Sections 3-4:

```
After '4711 ': expected ' ' or Plus or end of text
```

This message is the result of several failures caused by `"#"` appearing after `"4711 "`. First, `Space()` failed to detect one more blank, which is reflected by `expected ' '`. Then, `Sum_0()` failed to find `Plus` there. Finally, `Input()` failed to see end of the text.

If you think the information about expected blank is uninteresting, it is possible to write a semantic action for `Space` that erases its error information,

8.2 Recovery

Error recovery is about continuing after a syntax error. Compilers usually skip a faulty statement and continue with next one. To see how this is done using *Mouse*, suppose you want to modify the parser from Sections 3-4 so that it accepts a sequence of `Sum`s separated by semicolons, and skips faulty text up to the next semicolon. Such parser could be defined as follows:

```
Input   = Space (&_(Sum / Skip))+ !_ ;
Skip    = (!End _)+ End;
Sum     = Number (Plus Number)* End {sum} ~{error} ;
Number  = Digits Space {number} ;
Plus    = "+" Space ;
Digits  = [0-9]+ ;
Space   = " "* ;
End     = ";" Space / !_ ;
```

If `Sum` in the sub-expression `(Sum / Skip)` fails, `Skip` is invoked to skip everything up to the next semicolon or end of input. Note that this grammar uses twice the standard PEG idiom for "repeat `A` until `B`", namely "`(!B A)+ B`".

The action name {`error`} preceded by a tilde at the end of `Sum` identifies a semantic action `error()` to be called if `Sum` fails. (Otherwise semantic actions are called only upon success.) The action has access to the `Phrase` object that represents the failed `Sum` and contains information about the cause of the failure. The function of `error` is to print out that information before it is superseded by later failures. This can be coded as follows:

```
void error()
  {
    System.out.println(lhs().errMsg());
    lhs().errClear();
  }
```

The helper method `errMsg()` constructs a printable message from the error information contained in a `Phrase`. The helper method `errClear()` removes that information to make sure it will not be issued again.

References

1. Ford, B. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, September 2002.
2. Ford, B. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming* (October 2002).
3. 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.
4. Grimm, R. Rats! – an easily extensible parser generator. `http://www.cs.nyu.edu/rgrimm/xtc/rats.html`.
5. Grimm, R. Practical packrat parsing. Tech. Rep. TR2004-854, Dept. of Computer Science, New York University, March 2004.
6. Lucas, P. The structure of formula-translators. *ALGOL Bulletin Supplement 16* (September 1961), 1–27.
7. Redziejowski, R. R. Parsing Expression Grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae 79*, 3–4 (2007), 513–524.
8. Redziejowski, R. R. Some aspects of Parsing Expression Grammar. *Fundamenta Informaticae 85*, 1–4 (2008), 441–454.
9. Redziejowski, R. R. Applying classical concepts to Parsing Expression Grammar. *Fundamenta Informaticae 93*, 1–3 (2009), 325–336.
10. Redziejowski, R. R. Mouse: from Parsing Expressions to a practical parser, 2009. `http://www.romanredz.se/mouse/Mouse-1.1.manual.pdf`.