Homework 1: LL(1) parsing

This assignment was inspired by the IEEE Std 1003.1-2004 international standard for the Awk programming language. As noted in Austin Common Standards Revision Group Enhancement Request Numbers 79 and 80 (2005-12-13, listed in an unapproved draft of changes), the standard contains a typographical error and a grammatical ambiguity. The ambiguity was resolved in the followup discussion (2006-01-06) and the problem was corrected in POSIX 1003.1-2008.

Consider the grammar:

```
expr    ::= num | lvalue | incrop expr | expr incrop | expr binop expr | (expr)
lvalue  ::= $expr
incrop  ::= ++ | --
binop   ::= + | -
num     ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

where {expr,lvalue,incrop,binop,num} is the set of non-terminal symbols, expr is the start symbol, and {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, (,), $, ++, --} is the set of terminal symbols. The grammar generates a subset of the Awk expressions. The various operators have precedences as in Awk as corrected by discussions noted above, so that a field reference (using $) has the highest precedence, post-increment and post-decrement operators have the second highest precedence, the pre-increment and pre-decrement operators are third, addition and subtraction are fourth, and string concatenation (indicated by an empty binop) has the lowest precedence; and all binary operators are left associative.

Rewrite the grammar into a grammar which is LL(1), and use the rewritten grammar as the basis for implementing a recursive descent parser in Java. Your parser should use a greedy scanner: that is, whenever it sees the two characters ++ in a left-to-right scan, it should treat it as the ++ token, instead of as two adjacent + tokens. Except when distinguishing ++ from + +, and -- from - -, the scanner should ignore and discard tabs, spaces, newlines, and Awk comments. An Awk comment starts with # and continues up to (but not including) the next newline.

If the input can be parsed correctly, the parser should output the expression in postfix notation, with a space before each operand, and a newline after the entire expression. The output should use an underscore _ to denote string concatenation, ++_ and --_ to denote pre-increment and pre-decrement, and _++ and _-- to denote post-increment and post-decrement.

Standard testing platform

We will test your submission using the Java Standard Edition 6 installed on SEASnet. There is no need to run on older Java versions, or on other platforms, though you may well wish to debug on your personal computer.

Your implementation should compile cleanly, without any warnings.

To run Java SE 6 on SEASnet, prepend `/usr/local/cs/bin` to your `PATH`, e.g., by appending the following lines to your `$HOME/.profile` file if you use `bash` or `ksh`:

```
export PATH=/usr/local/cs/bin:$PATH
```

or the following line to your `$HOME/.login` file if you use `tcsh` or `csh`:

```
set path=(/usr/local/cs/bin $path)
```

The command `java -version` should output the following text:

```
java version "1.6.0_03"
Java(TM) SE Runtime Environment (build 1.6.0_03-b05)
Java HotSpot(TM) 64-Bit Server VM (build 1.6.0_03-b05, mixed mode)
```

Submit

Submit on paper the LL(1) grammar, the FIRST and FOLLOW sets for each nonterminal symbol, and the predictive parsing table. Argue that the grammar is LL(1). In your paper description, please use `F` instead of `$` to denote the field reference operator, so that you can continue to use `$` to denote end of input. If you prefer, you may submit your copy in PDF form rather than on paper; if so, call it `hw1-grammar.pdf`.

Submit your program electronically. Your main file should be called `Parse.java`, and if `Expression` is a file, possibly containing an Awk expression according to the above grammar, then:

```
java Parse < Expression
```

outputs:

- either the postfix version of the input followed by the line "Expression parsed successfully",
- or "Parse error in line <linenumber>".

where <linenumber> is the decimal line number of the line where the first invalid symbol was found.

For example, if the input is:

```
$1 +
(1 - ++$2) $# (a confusing comment)
3
```

then the output should be:

```
 1 $ 1 2 $ ++_ - + 3 $ _
Expression parsed successfully
```

Here's another example. If the input is:

```
$$1++++$2
```

then the output should be:

```
 1 $ $ _++ _++ 2 $ _
Expression parsed successfully
```

This second example is semantically incorrect, since you cannot apply postincrement twice to the same expression in Awk (as in C or C++), but you need not worry about this, as this assignment deals only with Awk syntax.

As this second example illustrates, the fact that post-increment and post-decrement operators have higher priority than pre-increment and pre-decrement is important even if there is a potential string-concatenation operator between the increment or decrement operators. For example, `$1++--$2` should be parsed as if it were `(($1)++)--$2`, not as if it were `($1)++)(--($2))` or `($1)(++(--($2)))`, and this is true even though the required parse is semantically invalid.

$Id: hw1.html,v 1.20 2011/03/28 03:36:43 eggert Exp $