```
/*

File:    SimpCompRecDescent.jpp
Author:  zerksis d. umrigar (zdu@acm.org)
Copyright (C) 1997 Zerksis D. Umrigar
Last Update Time-stamp: "97/06/27 20:42:09 umrigar"

This code is distributed under the terms of the GNU General Public License.
See the file COPYING with this distribution, or

        http://www.fsf.org/copyleft/gpl.html

THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

*/

package zdu.parsdemo;

/*
Straight-forward recursive-descent parser whose derivation from a
translation scheme is sketched below:

Parser for following translation scheme (terminals enclosed within single
quotes or all upper-cased; semantic actions within braces; If a symbol
S has a synthesized attribute s, then the symbol is shown as S(s).

program
  : stmts
  ;
stmts
  : assgnStmt
  | assgnStmt ';' stmts
  ;
assgnStmt
  : ID ':='  expr
  ;
expr
  : expr '+' term
  | expr '-' term
  | term
  ;
term
  : term '*' factor
  | term '/' factor
  | term 'div' factor
  | term 'mod' factor
  | factor
  ;
factor
  : '(' expr ')'
  | ID
  | NUM
  ;
*/

FF

/*
After removing left-recursion, to allow recursive-descent parsing:

program
  : stmts
  ;
stmts
  : assgnStmt stmtsRest
  ;
```

```
stmtsRest
  : ';' stmts
  | EMPTY
  ;
assgnStmt
  : ID ':=' expr
  ;
expr
  : term exprRest
  ;
exprRest
  : '+' term exprRest
  | '-' term exprRest
  | EMPTY
  ;
term
  : factor termRest
  ;
termRest
  : '*' factor termRest
  | '/' factor termRest
  | 'div' factor termRest
  | 'mod' factor termRest
  | EMPTY
  ;
factor
  : '(' expr ')'
  | ID
  | NUM
  ;
*/
FF

#define CALL(nonTerm)                          \
  do {                                  \
    call((NonTerm) prgSyms.get(#nonTerm), ruleN, __LINE__); \
    nonTerm();                          \
  } while (false)

#define MATCH(tok)                      \
  do {                                  \
    match((Terminal) prgSyms.get(#tok), ruleN, __LINE__);   \
  } while (false)

#define RET()                           \
  do {                                  \
    ret(ruleN, __LINE__); return;               \
  } while (false)

#define ACCEPT()                        \
  do {                                  \
    accept(ruleN, __LINE__); return;                \
  } while (false)

#define RULE(n)     int ruleN= n

FF

import zdu.parsdemo.NonTerm;
import zdu.parsdemo.ParseException;
import zdu.parsdemo.ParseResetException;
import zdu.parsdemo.RecParser;
import zdu.parsdemo.Scanner;
import zdu.parsdemo.SimpCompLL1Gram;
import zdu.parsdemo.Terminal;

```

```
133  import java.util.Hashtable;
134
135  class SimpCompRecDescent extends RecParser
136  {
137
138    SimpCompRecDescent(SimpCompLL1Gram grammar, Scanner scanner,
139                ParseDisplay parseDisplay) {
140      super(grammar, scanner, parseDisplay);
141
142      prgSyms.put("assgnStmt", grammar.assgnStmt);
143      prgSyms.put("expr", grammar.expr);
144      prgSyms.put("exprRest", grammar.exprRest);
145      prgSyms.put("factor", grammar.factor);
146      prgSyms.put("program", grammar.program);
147      prgSyms.put("stmts", grammar.stmts);
148      prgSyms.put("stmtsRest", grammar.stmtsRest);
149      prgSyms.put("term", grammar.term);
150      prgSyms.put("termRest", grammar.termRest);
151
152      prgSyms.put("'+'", grammar.ADD);
153      prgSyms.put("ASSGN", grammar.ASSGN);
154      prgSyms.put("DIV", grammar.DIV);
155      prgSyms.put("'/'", grammar.DIVIDE);
156      prgSyms.put("ID", grammar.ID);
157      prgSyms.put("'('", grammar.LPAREN);
158      prgSyms.put("MOD", grammar.MOD);
159      prgSyms.put("'*'", grammar.MULT);
160      prgSyms.put("NUM", grammar.NUM);
161      prgSyms.put("')'", grammar.RPAREN);
162      prgSyms.put("';'", grammar.SEMI);
163      prgSyms.put("'-'", grammar.SUB);
164
165
166    }
167
168    SimpCompRecDescent(SimpCompLL1Gram grammar, Scanner scanner) {
169      this(grammar, scanner, null);
170    }
171
172    protected final void parse() throws ParseException, ParseResetException
173    {
174      RULE(0);
175      CALL(program);
176      ACCEPT();
177    }
178
179    private void program() throws ParseException, ParseResetException
180    {
181      RULE(1);
182      CALL(stmts);
183    }
184
185    private void stmts() throws ParseException, ParseResetException
186    {
187      RULE(2);
188      CALL(assgnStmt);
189      CALL(stmtsRest);
190      RET(); /* stmts() */
191    }
192
193    private void stmtsRest() throws ParseException, ParseResetException
194    {
195      if (tok.getTokNum() == ';') {
196        RULE(3);
197        MATCH(';');
198        CALL(stmts);
```

```
199         RET(); /* stmtsRest() */
200       }
201       else {
202         RULE(4);
203         RET(); /* stmtsRest() */
204       }
205     }
206
207     private void assgnStmt() throws ParseException, ParseResetException
208     {
209       RULE(5);
210       MATCH(ID);
211       MATCH(ASSGN);
212       CALL(expr);
213       RET(); /* assgnStmt() */
214     }
215
216     private void expr() throws ParseException, ParseResetException
217     {
218       RULE(6);
219       CALL(term);
220       CALL(exprRest);
221       RET(); /* expr() */
222     }
223
224     private void exprRest() throws ParseException, ParseResetException
225     {
226       if (tok.getTokNum() == '+') {
227         RULE(7);
228         MATCH('+');
229         CALL(term);
230         CALL(exprRest);
231         RET(); /* exprRest() */
232       }
233       else if (tok.getTokNum() == '-') {
234         RULE(8);
235         MATCH('-');
236         CALL(term);
237         CALL(exprRest);
238         RET(); /* exprRest() */
239       }
240       else {
241         RULE(9);
242         RET(); /* exprRest() */
243       }
244     }
245
246     private void term() throws ParseException, ParseResetException
247     {
248       RULE(10);
249       CALL(factor);
250       CALL(termRest);
251       RET(); /* term() */
252     }
253
254     private void termRest() throws ParseException, ParseResetException
255     {
256       if (tok.getTokNum() == '*') {
257         RULE(11);
258         MATCH('*');
259         CALL(factor);
260         CALL(termRest);
261         RET(); /* termRest() */
262       }
263       else if (tok.getTokNum() == '/') {
264         RULE(12);
```

```
265         MATCH('/');
266         CALL(factor);
267         CALL(termRest);
268         RET(); /* termRest() */
269       }
270       else if (tok.getTokNum() == SimpCompScanner.DIV_TOK) {
271         RULE(13);
272         MATCH(DIV);
273         CALL(factor);
274         CALL(termRest);
275         RET(); /* termRest() */
276       }
277       else if (tok.getTokNum() == SimpCompScanner.MOD_TOK) {
278         RULE(14);
279         MATCH(MOD);
280         CALL(factor);
281         CALL(termRest);
282         RET(); /* termRest() */
283       }
284       else {
285         RULE(15);
286         RET(); /* termRest() */
287       }
288     }
289
290     private void factor() throws ParseException, ParseResetException
291     {
292       if (tok.getTokNum() == '(') {
293         RULE(16);
294         MATCH('(');
295         CALL(expr);
296         MATCH(')');
297         RET(); /* factor() */
298       }
299       else if (tok.getTokNum() == SimpCompScanner.ID_TOK) {
300         RULE(17);
301         MATCH(ID);
302         RET(); /* factor() */
303       }
304       else {
305         RULE(18);
306         String v= new String(scanner.lexemeText());
307         MATCH(NUM);
308         RET(); /* factor() */
309       }
310     }
311
312     private Hashtable prgSyms= new Hashtable();
313
314   }
315
316
317
```