```
 1 // Brandon Chavez, CSCI 450 - Fall 2018, Professor Nelson
 2 /* parse.c:  This is where the parser should be built. */
 3 // Rated "E", for "Even You!".
 4 #define TRUE 1
 5 #define FALSE 0
 6 /* Get the standard definitions and includes. */
 7 #include "defs.h"
 8
 9 /* External Variables. */
10 #include "global.h"
11
12 /* static "local" variables. */
13 token currentToken;
14 id_rec* symbolTable = NULL;
15 int stringStorageMade = FALSE;
16
17 // Functions for interacting with the symbol table.
18 void lookupSymbolA(void);
19 void lookupSymbolB(void);
20 void addSymbol(void);
21 /* Functions for the rules. */
22 void printToken(FILE* stream, token t);
23 void match(token t);
24 void goal(void);
25 void var_decl(void);
26 void id_list(void);
27 void statements(void);
28 void statement(void);
29 void read_list(void);
30 void write_list(void);
31 void write_elem(void);
32 void expr(void);
33 void add_op(void);
34 void primary(void);
35
36
37 void parse (void)
38 {
39     currentToken = scanner();
40     if(list_src)
41     {
42         while(currentToken != SCANEOF)
43         {
44             printToken(stdout, currentToken);
45             currentToken = scanner();
46             printf("\n");
47         }
48         printToken(stdout, currentToken);
49     }else
50     {
```

```
51          goal();
52          if(currentToken != SCANEOF)
53          {
54              fprintf(stderr, "Syntax Error: Expected an EOF
     token!\n");
55          }else
56          {
57              fprintf(stdout, "Parsing successful! Gooooo
   Joooooe!\n");
58          }
59      }
60  }
61  //Prints the text value of a token based on enumeration
   value.
62  void printToken(FILE* stream, token t)
63  {
64      switch(t)
65      {
66          case ID:
67              fprintf(stream, "ID: %s", mostRecentToken);
68              break;
69          case STRING:
70              fprintf(stream, "STRING: %s", mostRecentToken)
   ;
71              break;
72          case CONST:
73              fprintf(stream, "CONST: %s", mostRecentToken);
74              break;
75          case END:
76              fprintf(stream, "end");
77              break;
78          case READ:
79              fprintf(stream, "read");
80              break;
81          case BEGIN:
82              fprintf(stream, "begin");
83              break;
84          case WRITE:
85              fprintf(stream, "write");
86              break;
87          case INTEGER:
88              fprintf(stream, "integer");
89              break;
90          case PROGRAM:
91              fprintf(stream, "program");
92              break;
93          case WRITELN:
94              fprintf(stream, "writeln");
95              break;
96          case VARIABLE:
```

```
 97             fprintf(stream, "variable");
 98             break;
 99         case ASSIGN:
100             fprintf(stream, "assign");
101             break;
102         case PLUS:
103             fprintf(stream, "PLUS");
104             break;
105         case MINUS:
106             fprintf(stream, "MINUS");
107             break;
108         case LPAREN:
109             fprintf(stream, "LEFT PARENTHESIS");
110             break;
111         case RPAREN:
112             fprintf(stream, "RIGHT PARENTHESIS");
113             break;
114         case SEMI:
115             fprintf(stream, "SEMICOLON");
116             break;
117         case COMMA:
118             fprintf(stream, "COMMA");
119             break;
120         case COLON:
121             fprintf(stream, "COLON");
122             break;
123         case PERIOD:
124             fprintf(stream, "PERIOD");
125             break;
126         case SCANEOF:
127             fprintf(stream, "END OF FILE.");
128             break;
129     }
130 }
131 //Determines if a symbol already exists in the Symbol
    Table, alerting the user if a double declaration has
    occurred,
132 //and simply adding the symbol's ID to the table otherwise
    .
133 void lookupSymbolA(void)
134 {
135     id_rec* temp;
136     //The mostRecentToken is assumed to be an ID at the
    time that lookupSymbol is called.
137     temp = find_id(symbolTable, mostRecentToken);
138     if(temp == NULL)
139     {
140         addSymbol();
141     }else if(!(strcmp(temp->id, symbolTable->id))) //It is
     assumed here that the root of the tree is always going to
```

```
141  be the program ID.
142     {
143         fprintf(stderr, "Semantic Error: Declaration of
    variable using program ID, %s.\n", symbolTable->id);
144         exit(EXIT_FAILURE);
145     }
146     else
147     {
148         fprintf(stderr, "Semantic Error: Double
    declaration of variable ID, %s.\n", mostRecentToken);
149         exit(EXIT_FAILURE);
150     }
151 }
152 //Alternate lookup function that prevents using an
    undeclared variable or the program ID inappropriately.
153 void lookupSymbolB(void)
154 {
155     id_rec* temp;
156     temp = find_id(symbolTable, mostRecentToken);
157     if(temp == NULL || (!strcmp(temp->id, symbolTable->id)
    ))
158     {
159         fprintf(stderr, "Semantic Error: The variable you
    are trying to assign a value to is not initialized, "
160                         "or you are attempting to assign
    to the program ID.\n");
161         exit(EXIT_FAILURE);
162     }
163 }
164 void addSymbol(void)
165 {
166     id_rec* temp = (id_rec*) malloc(sizeof(id_rec));
167     temp->id = strdup(mostRecentToken);
168     // Set fields here if you need them! Can't think of
    anything we'd need for now, since ATL0 only has integer
    type variables.
169     insert_id(&symbolTable, temp);
170 }
171 //Small helper function which simply verifies that a given
     token matches another
172 //desired or expected token.
173 void match(token t)
174 {
175     if(currentToken == t)
176     {
177         currentToken = scanner();
178     }else
179     {
180         fprintf(stderr, "Syntax Error: Expected token, ");
181         printToken(stderr, t);
```

```
182         fprintf(stderr, ", but received: ");
183         printToken(stderr, currentToken);
184         fprintf(stderr, ".\n");
185         exit(EXIT_FAILURE);
186     }
187 }
188 //Non-Terminal Processing Functions.
189 //goal is the "start" symbol, from which further rule
    procedures are applied.
190 void goal(void)
191 {
192     char firstID[50];
193     char secondID[50];
194 
195     match(PROGRAM);
196     //Add the program ID to the Symbol Table.
197     addSymbol();
198     //Semantic check to ensure the ID's at the beginning
    and end of the program match.
199     strcpy(firstID, mostRecentToken);
200     match(ID);
201     match(SEMI);
202     var_decl();
203     match(BEGIN);
204     statements();
205     match(END);
206     strcpy(secondID, mostRecentToken);
207     match(ID);
208     if(strcmp(firstID, secondID))
209     {
210         fprintf(stderr, "Semantic Error: program ID's at
    the beginning and end do not match.\n");
211         exit(EXIT_FAILURE);
212     }
213     match(PERIOD);
214     //Stops the hc upon finishing the program.
215     generate("halt");
216 }
217 void var_decl(void)
218 {
219     match(VARIABLE);
220     id_list();
221     match(COLON);
222     match(INTEGER);
223     match(SEMI);
224 }
225 void id_list(void)
226 {
227     lookupSymbolA();
228     //Generate assembly instructions for variable
```

```
228 declaration.
229     label("%s", mostRecentToken);
230     generate(".bss 1");
231     match(ID);
232     while(currentToken == COMMA)
233     {
234         match(COMMA);
235         lookupSymbolA();
236         label("%s", mostRecentToken);
237         generate(".bss 1");
238         match(ID);
239     }
240 }
241 void statements(void)
242 {
243     statement();
244     match(SEMI);
245     while(currentToken == ID || currentToken == READ ||
246        currentToken == WRITE || currentToken == WRITELN)
247     {
248         statement();
249         match(SEMI);
250     }
251 }
252 void statement(void)
253 {
254     switch(currentToken)
255     {
256         case ID: {
257             lookupSymbolB();
258             //Store ID of the variable we're about to
    assign to.
259             char varID[strlen(mostRecentToken) + 1];
260             strcpy(varID, mostRecentToken);
261             match(ID);
262             match(ASSIGN);
263             expr();
264             //Assign value to the variable stored.
265             generate("stor 0,%s", varID);
266             break;
267         }
268         case READ: {
269             match(READ);
270             match(LPAREN);
271             read_list();
272             match(RPAREN);
273             break;
274         }
275         case WRITE: {
276             match(WRITE);
```

```
277             match(LPAREN);
278             write_list();
279             match(RPAREN);
280             break;
281         }
282         case WRITELN: {
283             match(WRITELN);
284             match(LPAREN);
285             match(RPAREN);
286             generate("csp wln");
287             break;
288         }
289         default: {
290             fprintf(stderr, "Invalid token for beginning
    of a statement.\n");
291             exit(EXIT_FAILURE);
292         }
293     }
294 }
295 void read_list(void)
296 {
297     lookupSymbolB();
298     generate("ladr 0,%s", mostRecentToken);
299     generate("csp rdi");
300     match(ID);
301     while(currentToken == SEMI)
302     {
303         match(SEMI);
304         lookupSymbolB();
305         generate("ladr 0,%s", mostRecentToken);
306         generate("csp rdi");
307         match(ID);
308     }
309 }
310 void write_list(void)
311 {
312     write_elem();
313     while(currentToken == SEMI)
314     {
315         match(SEMI);
316         write_elem();
317     }
318 }
319 void write_elem(void)
320 {
321     if(currentToken == STRING)
322     {
323         if(!stringStorageMade)
324         {
325             label("tmp$stor");
```

```
326             generate(".bss %d", 100);
327             stringStorageMade = TRUE;
328         }
329         generate("ladr 0,tmp$stor");
330         generate("movs %s", mostRecentToken);
331         generate("pint 0");
332         generate("csp wrs");
333         match(STRING);
334     }else
335     {
336         expr();
337         generate("pint 10");
338         generate("csp wri");
339     }
340 }
341 //This effectively combines the derivations for expr and
    add_op into one expr rule.
342 void expr(void)
343 {
344     primary();
345     while(currentToken == PLUS || currentToken == MINUS)
346     {
347         char op[4];
348         if(currentToken == PLUS)
349         {
350             strcpy(op, "add");
351         }else
352         {
353             strcpy(op, "sub");
354         }
355         match(currentToken);
356         primary();
357         //Add or subtract depending on what operation we
    found earlier.
358         generate("%s", op);
359     }
360     //generate("csp wri");
361 }
362 void primary(void)
363 {
364     switch(currentToken)
365     {
366         case LPAREN:
367             match(LPAREN);
368             expr();
369             match(RPAREN);
370             break;
371         case ID:
372             lookupSymbolB();
373             //Load variable onto stack.
```

```
374             generate("load 0,%s", mostRecentToken);
375             match(ID);
376             break;
377         case CONST:
378             //Load integer constant onto stack.
379             generate("pint %s", mostRecentToken);
380             match(CONST);
381             break;
382         default:
383             fprintf(stderr, "Invalid token for beginning
    of a primary.\n");
384             exit(EXIT_FAILURE);
385 
386     }
387 }
388 
```