```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the io module that serves as the input for the scanner
 */
#include "io.h"
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>

unsigned pclineno = 1;      /* current line number */

int pcerror(const char *format, ...) {
    va_list args;

    /* va print to the error console */
    va_start(args, format);
    vfprintf(stderr, format, args);
    va_end(args);
    return 0;
}

char pcgetc(FILE *fp) {
    return fgetc(fp);
}

void pcungetc(char c, FILE *fp) {
    ungetc(c, fp);
}
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the io module that serves as the input for the scanner
 */

#ifndef IO_H
#define IO_H

#include <stdio.h>

extern unsigned pclineno;   /* current line number */

/*
Prints out an error message to the error console.
@see printf.
@return always return 0
*/
int pcerror(const char *format, ...);

/*
Gets the next character from the FILE.

@param fp the FILE pointer
@return next character in the FILE
*/
char pcgetc(FILE *fp);

/*
Puts a character back onto the FILE.

@param c the character to put back into the FILE
@param fp the FILE pointer
*/
void pcungetc(char c, FILE *fp);

#endif /* IO_H */
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * This is the program that creates the abstract sytax tree
    printout
 5  */
 6 #include "ast.h"
 7 #include "io.h"
 8 #include <stdlib.h>
 9
10 AST *astroot = NULL;
11
12 AST* AST_initialize(ASTnode node) {
13 AST *ast;
14
15     if (!(ast = malloc(sizeof(*ast)))) {
16         pcerror("Out of memory.\n");
17         return NULL;
18     }
19
20     ast->node = node;
21     ast->name = NULL;
22     ast->sym = eofsym;
23     ast->val.ival = 0;
24     ast->head = NULL;
25     ast->tail = NULL;
26
27     return ast;
28 }
29
30 int AST_addchild(AST *root, AST* child) {
31 ASTchild *cur;
32
33     if (!(cur = malloc(sizeof(*cur)))) {
34         return pcerror("Out of memory.\n");
35     }
36
37     cur->ast = child;
38     cur->next = NULL;
39
40     /* see if we have any children yet, and initialize if
   we don't */
41     if (!root->head) {
42         root->head = cur;
43         root->tail = cur;
44
45         return 1;
46     }
47
48     /* update the tail */
```

```
49      root->tail->next = cur;
50      root->tail = cur;
51      return 1;
52  }
53
54  void AST_cleanup(AST **root) {
55  AST *ast;
56  ASTchild *cur;
57
58      ast = *root;
59
60      /* clean up all our children, left-to-right, starting
    at the deepest child */
61      while ((cur = ast->head)) {
62          AST_cleanup(&(cur->ast));
63          ast->head = cur->next;
64
65          /* cleanup the actual child */
66          cur->next = NULL;
67          free(cur);
68      }
69
70      if (ast->name) free(ast->name);
71      ast->name = NULL;
72      ast->head = NULL;
73      ast->tail = NULL;
74
75      free(ast);
76      *root = NULL;
77  }
78
79  const char *AST_nodestr(ASTnode node) {
80      switch (node) {
81          /* End-of-Tokens */
82          case eofasm: return "eof";
83
84          /* Operators */
85          case addasm: return "add";
86          case multasm: return "mult";
87
88          /* Scopes */
89          case programasm: return "program";
90          case procedureasm: return "procedure";
91          case functionasm: return "function";
92          case paramasm: return "param";
93          case statementasm: return "statement";
94          case proccallasm: return "proccall";
95          case funccallasm: return "funccall";
96
97          /* Expressions */
```

```
 98         case exprasm: return "expr";
 99         case simexprasm: return "simexpr";
100         case termasm: return "term";
101         case factorasm: return "factor";
102 
103         /* Boolean operators */
104         case relasm: return "rel";
105         case notasm: return "not";
106 
107         /* Punctuation */
108         case assignasm: return "assign";
109         case dotdotasm: return "dotdot";
110 
111         /* Control flow */
112         case ifasm: return "if";
113         case whileasm: return "while";
114 
115         /* Variables */
116         case idasm: return "id";
117         case arrayasm: return "array";
118         case ofasm: return "of";
119         case charasm: return "char";
120         case stringasm: return "string";
121         case integerasm: return "integer";
122         case realasm: return "real";
123         case varasm: return "var";
124 
125         /* Constants */
126         case valasm: return "val";
127         case constasm: return "const";
128 
129         /* Built-in functions */
130         case chrasm: return "chr";
131         case ordasm: return "ord";
132         case readasm: return "read";
133         case readlnasm: return "readln";
134         case writeasm: return "write";
135         case writelnasm: return "writeln";
136 
137         /* Number of syms */
138         case numasms: return "numasms";
139 
140         default: return "ERR";
141     }
142 
143     return "ERR";
144 }
145 
146 void AST_print_internal(AST *root, FILE *fp, int depth) {
147     int i = depth;
```

```
148     char str[1024];
149     char *c = str;
150     ASTchild *cur = root->head;
151 
152     while (i--) *c++ = '\t';
153 
154     if (root->name) {
155         if (root->val.ival) snprintf(c, 1023-depth, "[%s
    name:%s val:Y]\n", AST_nodestr(root->node), root->name);
156         else snprintf(c, 1023-depth, "[%s name:%s]\n",
    AST_nodestr(root->node), root->name);
157     } else {
158         if (root->val.ival) snprintf(c, 1023-depth, "[%s
    val:Y]\n", AST_nodestr(root->node));
159         else snprintf(c, 1023-depth, "[%s]\n", AST_nodestr
    (root->node));
160     }
161 
162     /* print to file */
163     fprintf(fp, "%s", str);
164 
165     /* print children in order */
166     while (cur) {
167         AST_print_internal(cur->ast, fp, depth+1);
168         cur = cur->next;
169     }
170 }
171 
172 void AST_print(AST *root, FILE *fp) {
173     AST_print_internal(root, fp, 0);
174 }
```

```
 1 /*
 2 * @author Derek Trom
 3 * @author Elena Corpus
 4         * This is the program that creates the abstract
   sytax tree printout
 5 */
 6 #ifndef AST_H
 7 #define AST_H
 8
 9 #include "tokens.h"
10 #include <stdio.h>
11
12 typedef enum {
13 /* End-of-Tokens */
14     eofasm = 0,
15
16     /* Operators */
17     addasm,
18     multasm,
19
20     /* Scopes */
21     programasm,
22     procedureasm,
23     functionasm,
24     paramasm,
25     statementasm,
26     proccallasm,
27     funccallasm,
28
29     /* Expressions */
30     exprasm,
31     simexprasm,
32     termasm,
33     factorasm,
34
35     /* Boolean operators */
36     relasm,
37     notasm,
38
39     /* Punctuation */
40     assignasm,
41     dotdotasm,
42
43     /* Control flow */
44     ifasm,
45     whileasm,
46
47     /* Variables */
48     idasm,
49     arrayasm,
```

```
50      ofasm,
51      charasm,
52      stringasm,
53      integerasm,
54      realasm,
55      varasm,
56
57      /* Constants */
58      valasm,
59      constasm,
60
61      /* Built-in functions */
62      chrasm,
63      ordasm,
64      readasm,
65      readlnasm,
66      writeasm,
67      writelnasm,
68
69      /* Number of syms */
70      numasms
71  } ASTnode;
72
73  struct ASTchild;
74  typedef struct AST {
75      ASTnode             node;   /* node type */
76      char                *name;  /* name in the symbol table
     */
77      pcsym               sym;    /* symbol */
78      symval              val;    /* value */
79      struct ASTchild     *head;  /* left-most child */
80      struct ASTchild     *tail;  /* right-most child */
81  } AST;
82
83  typedef struct ASTchild {
84      AST*            ast;    /* value of the child */
85      struct ASTchild *next;  /* next child, left-to-right */
86  } ASTchild;
87
88  /* Our global AST */
89  extern AST *astroot;
90
91  /* Initializes an AST for use.
92
93  @param node the type of AST
94  @return memory allocated AST
95  */
96  AST* AST_initialize(ASTnode node);
97
98  /* Adds a child to the AST, in left-to-right order.
```

```
 99
100 @param child the AST to add
101 @return 1 on success; 0 otherwise
102 */
103 int AST_addchild(AST *root, AST *child);
104
105 /* Cleans up the memory for a given AST.
106
107 @param root the AST to cleanup
108 */
109 void AST_cleanup(AST **root);
110
111 /* Print an AST tree to the given file.
112
113 @param fp the file pointer
114 */
115 void AST_print(AST *root, FILE *fp);
116
117 #endif /* AST_H */
```

```
 1 PROJNAME = mini-pascal-compiler
 2 YYNAME = yy-mini-pascal-compiler
 3 CC = gcc
 4 CFLAGS =
 5 YYCFLAGS = -DYYCOMPILE
 6 YYOBJ = $(YYNAME).tab.o lex.yy.o
 7 LEX = scanner.l
 8 PARSE = parser.y
 9 PARSEFLAGS = -v -d
10 REMOVEFILES = parser.tab.* lex.yy.* $(PROJNAME) $(YYNAME
   ) *.s *.output *.o
11 SOURCES = compiler.c io.c scanner.c symtab.c tokens.c
   parser.c ast.c
12 YYSOURCES = compiler.c parser.tab.c lex.yy.c
13
14 UNAME_S := $(shell uname -s)
15 ifeq ($(UNAME_S),Linux)
16     YYCFLAGS += -lfl
17 endif
18 ifeq ($(UNAME_S),Darwin)
19     YYCFLAGS += -ll
20 endif
21
22 hand: $(PROJNAME)
23
24 yy: $(YYNAME)
25
26 all: $(PROJNAME) $(YYNAME)
27
28 debug: debughand debugyy
29
30 debughand: CFLAGS += -DDEBUG
31 debughand: $(PROJNAME)
32
33 debugyy: YYCFLAGS += -DDEBUG
34 debugyy: $(YYNAME)
35
36 $(PROJNAME):
37     $(CC) $(SOURCES) $(CFLAGS) -o $@
38
39 $(YYNAME): $(YYOBJ)
40     $(CC) $(YYSOURCES) $(YYCFLAGS) -o $@
41
42 $(YYNAME).tab.o: $(PARSE)
43     bison $(PARSEFLAGS) $(PARSE)
44
45 lex.yy.o: $(LEX)
46     flex $(LEX)
47
48 clean:
```

```
49      rm -f $(REMOVEFILES)
50
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the parser program that recursively parses the scanned input file.
 */
#include "parser.h"
#include "symtab.h"
#include "ast.h"
#include "io.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int pcp_block();
int pcp_statement_part();
int pcp_application();
int pcp_constant_definition();
int pcp_expression();

FILE *fp = NULL;
pctoken *lasttoken = NULL;
pctoken *token = NULL;
pctoken *nexttoken = NULL;

#define NEXTTOKEN() if (!pcp_next()) return 0
#define ADDTOKEN(TAIL, TOKEN) if (!(TAIL = tokenlist_add(TAIL, TOKEN))) return 0
#define EXPECT(SYM) if (!pcp_expect(SYM)) return 0
#define EXPECT2(SYM1, SYM2) if (!pcp_expect2(SYM1, SYM2)) return 0

typedef struct pctokenlist {
    pctoken *token;
    struct pctokenlist *next;
} pctokenlist;


/* Updates the tokens */
int pcp_next() {
    /* get the next token */
    lasttoken = token;
    token = nexttoken;
    nexttoken = pcgettoken(fp);

    if (!token) {
        return pcerror("Unexpected end of tokens.\n");
    }

    /* print the line from the scanner */
```

```
48      printf("< %s ", pcsymstr[token->sym]);
49
50      if (token->sym == idsym) {
51          printf(", %s ", token->val.id);
52      } else if (token->sym == integernosym) {
53          printf(", %d ", token->val.ival);
54      } else if (token->sym == realnosym) {
55          printf(", %f ", token->val.rval);
56      } else if (token->sym == stringvalsym) {
57          printf(", %s ", token->val.str);
58      } else if (token->sym == charvalsym) {
59          printf(", %c ", token->val.cval);
60      }
61
62      printf(">\n");
63      return 1;
64  }
65
66  /* Accepts a given symbol and skips the next symbol if a
    match is found.
67
68  @param sym the symbol to match against
69  @return 1 on success; 0 otherwise
70  */
71  int pcp_accept(pcsym sym) {
72      if (token->sym == sym) {
73          return pcp_next() != 0;
74      }
75
76      return 0;
77  }
78
79  /* Forces a specific symbol to be found.
80
81  @param sym the symbol to match against
82  @return 1 on success; 0 otherwise
83  */
84  int pcp_expect(pcsym sym) {
85      if (pcp_accept(sym)) {
86          return 1;
87      }
88
89      return pcerror("[%u] pcp_expect: Unexpected symbol: %s
    vs %s\n", token->lineno, token->val, pcsymstr[sym]);
90  }
91
92  /* Forces two specific symbols to be found in sequence.
93
94  @param sym1 the first symbol to match against
95  @param sym2 the second symbol to match against
```

```
 96 @return 1 on success; 0 otherwise
 97 */
 98 int pcp_expect2(pcsym sym1, pcsym sym2) {
 99 pcsym tmpsym = token->sym;
100
101     if (pcp_accept(sym1)) {
102         if (pcp_accept(sym2)) {
103             return 1;
104         }
105
106         return pcerror("[%u] pcp_expect: Unexpected end to
     symbol sequence: %s %s vs %s %s\n",
107             token->lineno, pcsymstr[tmpsym], pcsymstr[
    token->sym], pcsymstr[sym1], pcsymstr[sym2]);
108     }
109
110     return pcerror("[%u] pcp_expect: Unexpected start to
    symbol sequence: %s %s vs %s %s\n",
111         token->lineno, pcsymstr[tmpsym], pcsymstr[
    nexttoken->sym], pcsymstr[sym1], pcsymstr[sym2]);
112 }
113
114 /* Converts a symbol to a symtype used to store in the
    symbol table.
115
116 @param sym the symbol to check for type
117 @param type return value of the type
118 @return 1 on success; 0 otherwise
119 */
120 int sym_to_type(pcsym sym, symtype *type) {
121     if (sym == integersym) *type = integertype;
122     else if (sym == realsym) *type = realtype;
123     else if (sym == stringsym) *type = stringtype;
124     else if (sym == charsym) *type = chartype;
125     else return pcerror("Unknown type. Arrays and custom
    types not yet supported.\n");
126
127     return 1;
128 }
129
130 /* Adds a pctoken to the end of our list.
131
132 @param tail the tail of our list
133 @param token the token to add
134 @return the new tail pointer
135 */
136 pctokenlist *
137 tokenlist_add(pctokenlist *tail, pctoken *token) {
138     pctokenlist *next;
139
```

```
140     if (!(next = malloc(sizeof(*next)))) {
141         pcerror("Out of memory.\n");
142         return NULL;
143     }
144     next->token = token;
145     next->next = NULL;
146 
147     tail->next = next;
148     return next;
149 }
150 
151 pctoken *
152 pcp_const_no_id(symtype *type) {
153     if (token->sym == integernosym)      *type =
    integertype;
154     else if (token->sym == realnosym)    *type = realtype;
155     else if (token->sym == charvalsym)   *type = chartype;
156     else if (token->sym == stringvalsym) *type =
    stringtype;
157     else return NULL;
158 
159     return token;
160 }
161 
162 /* Ensures the the ord() function is called correctly.
163 
164 @return 1 on success; 0 otherwise
165 */
166 int pcp_ord(AST *ast) {
167     AST *astord;
168 
169     printf("********ENTERING ord********\n");
170 
171     EXPECT(lparensym);
172 
173     astord = AST_initialize(ordasm);
174     AST_addchild(ast, astord);
175 
176     if (!pcp_expression(astord)) return 0;
177 
178     EXPECT(rparensym);
179 
180     return 1;
181 }
182 
183 /* Ensures the the chr() function is called correctly.
184 
185 @return 1 on success; 0 otherwise
186 */
187 int pcp_chr(AST *ast) {
```

```
188     AST *astchr;
189
190     printf("********ENTERING chr********\n");
191
192     EXPECT(lparensym);
193
194     astchr = AST_initialize(chrasm);
195     AST_addchild(ast, astchr);
196
197     if (!pcp_expression(astchr)) return 0;
198
199     EXPECT(rparensym);
200
201     return 1;
202 }
203
204 int pcp_factor(AST *ast) {
205     AST *astfactor;
206     AST *astother;
207     pctoken *ltoken;
208
209     printf("********ENTERED factor********\n");
210
211     astfactor = AST_initialize(factorasm);
212     AST_addchild(ast, astfactor);
213
214     if (pcp_accept(notsym)) {
215         astother = AST_initialize(notasm);
216         AST_addchild(astfactor, astother);
217         return pcp_factor(astfactor);
218     }
219
220     if (pcp_accept(idsym)) {
221         ltoken = lasttoken;
222         if (pcp_accept(lparensym)) {
223             return pcp_application(astfactor, ltoken);
224         } if (pcp_accept(lbracksym)) {
225             return pcerror("Arrays not yet supported.\n");
226         } else {
227             astother = AST_initialize(idasm);
228             astother->name = strdup(ltoken->val.id);
229             AST_addchild(astfactor, astother);
230         }
231
232         return 1;
233     }
234
235     if (pcp_accept(ordsym)) {
236         return pcp_ord(astfactor);
237     }
```

```
238
239     if (pcp_accept(chrsym)) {
240         return pcp_chr(astfactor);
241     }
242
243     if (pcp_accept(lparensym)) {
244         if (!pcp_expression(astfactor)) return 0;
245
246         pcp_expect(rparensym);
247
248         return 1;
249     }
250
251     /* see if we have an inline 'constant' value */
252     if (token->sym == integernosym || token->sym ==
    realnosym || token->sym == stringvalsym || token->sym ==
    charvalsym) {
253         astother = AST_initialize(valasm);
254         astother->sym = token->sym;
255
256         if (token->sym == stringvalsym) {
257             astother->val.str = strdup(token->val.str);
258         } else {
259             astother->val = token->val;
260         }
261
262         AST_addchild(astfactor, astother);
263         NEXTTOKEN();
264         return 1;
265     }
266
267     /* failed all our branches */
268     return 0;
269 }
270
271 int pcp_term(AST *ast) {
272     AST *astterm;
273     AST *astmult;
274
275     printf("********ENTERED term********\n");
276
277     astterm = AST_initialize(termasm);
278     AST_addchild(ast, astterm);
279
280     if (!pcp_factor(astterm)) return 0;
281
282     /* keep doing all the multiplicitive arithmetic */
283     while (pcp_accept(multsym) || pcp_accept(idivsym) ||
    pcp_accept(divsym) || pcp_accept(andsym)) {
284         astmult = AST_initialize(multasm);
```

```
285         astmult->sym = lasttoken->sym;
286         AST_addchild(astterm, astmult);
287 
288         if (!pcp_factor(astterm)) return 0;
289     }
290 
291     return 1;
292 }
293 
294 int pcp_simple_expression(AST *ast) {
295     AST *astsimexpr;
296     AST *astaddasm;
297 
298     printf("********ENTERED simple_expression********\n");
299 
300     astsimexpr = AST_initialize(simexprasm);
301     AST_addchild(ast, astsimexpr);
302 
303     if (!pcp_term(astsimexpr)) return 0;
304 
305     /* keep doing all the additional arithemetic */
306     while (pcp_accept(addsym) || pcp_accept(minussym) ||
    pcp_accept(orsym)) {
307         astaddasm = AST_initialize(addasm);
308         astaddasm->sym = lasttoken->sym;
309         AST_addchild(astsimexpr, astaddasm);
310 
311         if (!pcp_term(astsimexpr)) return 0;
312     }
313 
314     /* skip next token */
315     /*NEXTTOKEN();*/
316     return 1;
317 }
318 
319 int pcp_expression(AST *ast) {
320     AST *astexpr;
321     AST *astrel;
322 
323     printf("********ENTERED expression********\n");
324 
325     astexpr = AST_initialize(exprasm);
326     AST_addchild(ast, astexpr);
327 
328     if (!pcp_simple_expression(astexpr)) return 0;
329 
330     /* see if this is relational */
331     if (pcp_accept(eqsym) || pcp_accept(neqsym) ||
    pcp_accept(ltsym) || pcp_accept(ltesym) || pcp_accept(
    gtesym) || pcp_accept(gtsym)) {
```

```
332         astrel = AST_initialize(relasm);
333         astrel->sym = lasttoken->sym;
334         AST_addchild(astexpr, astrel);
335         return pcp_simple_expression(astexpr);
336     }
337 
338     return 1;
339 }
340 
341 /*int
342 pcp_for() {
343     NEXTTOKEN();
344     if (token->sym != idsym) return pcp_error("Expected ID
    .");
345 
346     NEXTTOKEN();
347     if (token->sym != assignsym) return pcp_error("
    Expected ':='.");
348 
349     NEXTTOKEN();
350     if (!pcp_expression()) return 0;
351 
352     NEXTTOKEN();
353     if (token->sym != tosym && token->sym != downtosym)
    return pcp_error("Expected to or downto.");
354 
355     NEXTTOKEN();
356     if (!pcp_expression()) return 0;
357 
358     NEXTTOKEN();
359     if (token->sym != dosym) return pcp_error("Expected do
    .");
360 
361     NEXTTOKEN();
362     return pcp_statement_part();
363 }*/
364 
365 int pcp_while(AST *ast) {
366     AST *astwhile;
367 
368     printf("********ENTERED while********\n");
369 
370     astwhile = AST_initialize(whileasm);
371     AST_addchild(ast, astwhile);
372 
373     if (!pcp_expression(astwhile)) return 0;
374 
375     EXPECT(dosym);
376 
377     return pcp_statement_part(astwhile);
```

```
378 }
379
380 int pcp_if(AST *ast) {
381     AST *astif;
382
383     printf("********ENTERED if********\n");
384
385     astif = AST_initialize(ifasm);
386     AST_addchild(ast, astif);
387
388     if (!pcp_expression(astif)) return 0;
389
390     EXPECT(thensym);
391
392     if (!pcp_statement_part(astif)) return 0;
393
394     if (pcp_accept(elsesym)) {
395         return pcp_statement_part(astif);
396     }
397
398     return 1;
399 }
400
401 int pcp_write(AST *ast) {
402     AST *astwrite;
403
404     printf("********ENTERED write********\n");
405
406     EXPECT(lparensym);
407
408     astwrite = AST_initialize(writeasm);
409     AST_addchild(ast, astwrite);
410
411     if (!pcp_expression(astwrite)) return 0;
412
413     while (pcp_accept(commasym)) {
414         if (!pcp_expression(astwrite)) return 0;
415     }
416
417     EXPECT(rparensym);
418     return 1;
419 }
420
421 int pcp_read(AST *ast) {
422     AST *astread;
423     AST *astcur;
424
425     printf("********ENTERED read********\n");
426
427     EXPECT(lparensym);
```

```
428     EXPECT(idsym);
429
430     astread = AST_initialize(readasm);
431     AST_addchild(ast, astread);
432
433     astcur = AST_initialize(idasm);
434     astcur->name = strdup(lasttoken->val.id);
435     AST_addchild(astread, astcur);
436
437     while (pcp_accept(commasym)) {
438         EXPECT(idsym);
439         astcur = AST_initialize(idasm);
440         astcur->name = strdup(lasttoken->val.id);
441         AST_addchild(astread, astcur);
442     }
443
444     EXPECT(rparensym);
445     return 1;
446 }
447
448 int pcp_application(AST *ast, pctoken *ltoken) {
449     AST *astfunccall;
450     symentry *entry = pclookupsym(ltoken->val.id);
451     int params = 1;
452
453     printf("********ENTERED application********\n");
454
455     if (!entry || entry->type != functiontype) {
456         return pcerror("Undefined ID or unexpected type.\n
    ");
457     }
458
459     astfunccall = AST_initialize(funccallasm);
460     astfunccall->name = strdup(ltoken->val.id);
461     AST_addchild(ast, astfunccall);
462
463     if (!pcp_expression(astfunccall)) return 0;
464
465     while (pcp_accept(commasym)) {
466         if (!pcp_expression(astfunccall)) return 0;
467         ++params;
468     }
469
470     EXPECT(rparensym);
471     return 1;
472 }
473
474 int pcp_procedure_call(AST *ast, pctoken *ltoken) {
475     AST *astproccall;
476     symentry *entry = pclookupsym(ltoken->val.id);
```

```
477     int params = 1;
478 
479     printf("********ENTERED procedure_call********\n");
480 
481     if (!entry || entry->type != proceduretype) {
482         return pcerror("Undefined ID or unexpected type.\n
    ");
483     }
484 
485     astproccall = AST_initialize(proccallasm);
486     astproccall->name = strdup(ltoken->val.id);
487     AST_addchild(ast, astproccall);
488 
489     if (!pcp_expression(astproccall)) return 0;
490 
491     while (pcp_accept(commasym)) {
492         if (!pcp_expression(astproccall)) return 0;
493         ++params;
494     }
495 
496     EXPECT(rparensym);
497     return 1;
498 }
499 
500 int pcp_procedure_call_or_application(AST *ast, pctoken *
    ltoken) {
501     return pcp_procedure_call(ast, ltoken) ||
    pcp_application(ast, ltoken);
502 }
503 
504 int pcp_assign(AST *ast, pctoken *ltoken) {
505     AST *astassign;
506     AST *astlval;
507     symentry *entry = pclookupsym(ltoken->val.id);
508 
509     printf("********ENTERED assign********\n");
510 
511     if (!entry || (entry->type != integertype && entry->
    type != realtype && entry->type != chartype && entry->type
     != stringtype)) {
512         return pcerror("Undefined ID or unexpected type.\n
    ");
513     }
514 
515     astassign = AST_initialize(assignasm);
516     astassign->name = strdup(ltoken->val.id);
517     AST_addchild(ast, astassign);
518 
519     astlval = AST_initialize(idasm);
520     astlval->name = strdup(ltoken->val.id);
```

```
521     AST_addchild(astassign, astlval);
522 
523     return pcp_expression(astassign);
524 }
525 
526 int pcp_statement(AST *ast) {
527     int success = 0;
528 
529     printf("********ENTERED statement********\n");
530 
531     /* procedure/function call or assignment */
532     if (pcp_accept(idsym)) {
533         pctoken *oldtoken = lasttoken;
534 
535         if (pcp_accept(lparensym)) success = 
    pcp_procedure_call_or_application(ast, oldtoken);
536         else if (pcp_accept(assignsym)) success = 
    pcp_assign(ast, oldtoken);
537         else if (pcp_accept(lbracksym)) return pcerror("
    Arrays not yet supported.\n");
538         else return pcerror("Unexpected symbol.\n");
539     } else if (pcp_accept(readsym) || pcp_accept(readlnsym
    )) {
540         success = pcp_read(ast);
541     } else if (pcp_accept(writesym) || pcp_accept(
    writelnsym)) {
542         success = pcp_write(ast);
543     } else if (pcp_accept(ifsym)) {
544         success = pcp_if(ast);
545     } else if (pcp_accept(whilesym)) {
546         success = pcp_while(ast);
547     } /*else if (pcp_accept(forsym)) {
548         success = pcp_for();
549     }*/ else if (pcp_accept(beginsym)) {
550         success = pcp_statement_part(ast);
551     } else {
552         return pcerror("Unexpected statement.\n");
553     }
554 
555     if (!success) return 0;
556 
557     /*NEXTTOKEN();*/
558     EXPECT(semicolonsym);
559 
560     return 1;
561 }
562 
563 int pcp_statement_part(AST *ast) {
564     AST *aststatement;
565 
```

```
566     printf("********ENTERING statement_part********\n");
567 
568     EXPECT(beginsym);
569 
570     aststatement = AST_initialize(statementasm);
571     AST_addchild(ast, aststatement);
572 
573     /* go through all the statements until end */
574     while (!pcp_accept(endsym)) {
575         if (!pcp_statement(aststatement)) return 0;
576     }
577 
578     return 1;
579 }
580 
581 int pcp_formal_parameters(AST *ast) {
582     pctokenlist tokens = {NULL, NULL};
583     pctokenlist *tail = NULL;
584     pctokenlist *cur;
585     pctoken *val;
586     symtype type;
587     AST *astparam;
588     AST *astcur;
589 
590     printf("********ENTERING formal_parameters********\n"
    );
591 
592     EXPECT(idsym);
593     tokens.token = lasttoken;
594     tail = &tokens;
595 
596     astparam = AST_initialize(paramasm);
597     AST_addchild(ast, astparam);
598 
599     while (pcp_accept(commasym)) {
600         EXPECT(idsym);
601         ADDTOKEN(tail, lasttoken);
602     }
603 
604     EXPECT(colonsym);
605 
606     val = token;
607     if (!sym_to_type(val->sym, &type)) return 0;
608     NEXTTOKEN();
609 
610     /* add all the id's to the symbol table */
611     cur = &tokens;
612     while (cur != NULL) {
613         if (!pcaddparam(cur->token->val.id, type, cur->
    token->lineno)) return 0;
```

```
614
615         astcur = AST_initialize(idasm);
616         astcur->name = strdup(cur->token->val.id);
617         AST_addchild(astparam, astcur);
618
619         tail = cur;
620         cur = cur->next;
621         if (tail != &tokens) free(tail);
622     }
623
624     return 1;
625 }
626
627 int pcp_function_declaration(AST *ast) {
628     symtype type;
629     AST *astfunc;
630
631     printf("********ENTERING function_declaration********\
    n");
632
633     /* enter our new scope for the function */
634     EXPECT(idsym);
635     if (!pcenterscope(lasttoken->val.id, functiontype,
    lasttoken->lineno)) return 0;
636
637     astfunc = AST_initialize(functionasm);
638     astfunc->name = strdup(lasttoken->val.id);
639     AST_addchild(ast, astfunc);
640
641     EXPECT(lparensym);
642     if (!pcp_formal_parameters(astfunc)) return 0;
643
644     EXPECT(rparensym);
645     EXPECT(colonsym);
646
647     if (!sym_to_type(token->sym, &type)) return 0;
648
649     /* update our return type */
650     /*entry->returntype = type;*/
651
652     NEXTTOKEN();
653     EXPECT(semicolonsym);
654
655     if (!pcp_block(astfunc)) return 0;
656
657     /* leave the function scope */
658     return pcleavescope();
659 }
660
661 int pcp_procedure_declaration(AST *ast) {
```

```
662     AST *astproc;
663 
664     printf("********ENTERING procedure_declaration********
    \n");
665 
666     /* create our new scope for the new variables */
667     EXPECT(idsym);
668     if (!pcenterscope(lasttoken->val.id, proceduretype,
    lasttoken->lineno)) return 0;
669 
670     astproc = AST_initialize(procedureasm);
671     astproc->name = strdup(lasttoken->val.id);
672     AST_addchild(ast, astproc);
673 
674     EXPECT(lparensym);
675     if (!pcp_formal_parameters(astproc)) return 0;
676 
677     EXPECT(rparensym);
678     EXPECT(semicolonsym);
679 
680     if (!pcp_block(astproc)) return 0;
681 
682     /* leave the procedure scope */
683     return pcleavescope();
684 }
685 
686 int pcp_procedure_and_function_definition_part(AST *ast) {
687     printf("********ENTERING
    procedure_and_function_definition********\n");
688 
689     if (pcp_accept(proceduresym)) {
690         pcp_procedure_declaration(ast);
691     } else if (pcp_accept(functionsym)) {
692         pcp_function_declaration(ast);
693     } else return 1;
694 
695     /*NEXTTOKEN();*/
696     EXPECT(semicolonsym);
697 
698     return pcp_procedure_and_function_definition_part(ast
    );
699 }
700 
701 int pcp_variable_definition(AST *ast) {
702     pctokenlist tokens = {NULL, NULL};
703     pctokenlist *tail = NULL;
704     pctokenlist *cur;
705     pctoken *val;
706     symtype type;
707     AST *astcur;
```

```
708
709     printf("********ENTERING variable_definition********\n
    ");
710
711     EXPECT(idsym);
712     tokens.token = lasttoken;
713     tail = &tokens;
714
715     while (pcp_accept(commasym)) {
716         /* update the linked list */
717         EXPECT(idsym);
718         ADDTOKEN(tail, lasttoken);
719     }
720
721     EXPECT(colonsym);
722
723     val = token;
724     if (!sym_to_type(val->sym, &type)) return 0;
725     NEXTTOKEN();
726
727     /* add all the id's to the symbol table */
728     cur = &tokens;
729     while (cur != NULL) {
730         if (!pcaddsym(cur->token->val.id, type, (symval)0
    , 0, cur->token->lineno)) return 0;
731
732         /* add to the parse tree */
733         astcur = AST_initialize(idasm);
734         astcur->name = strdup(cur->token->val.id);
735         AST_addchild(ast, astcur);
736
737         tail = cur;
738         cur = cur->next;
739         if (tail != &tokens) free(tail);
740     }
741
742     EXPECT(semicolonsym);
743     if (token->sym == idsym) return
    pcp_variable_definition(ast);
744
745     return 1;
746 }
747
748 int pcp_variable_definition_part(AST *ast) {
749     AST *astvar;
750     printf("********ENTERING variable_definition********\n
    ");
751
752     if (!pcp_accept(varsym)) return 1;
753
```

```
754     astvar = AST_initialize(varasm);
755     AST_addchild(ast, astvar);
756 
757     return pcp_variable_definition(astvar);
758 }
759 
760 int pcp_constant_definition(AST *ast) {
761     pctokenlist tokens = {NULL, NULL};
762     pctokenlist *tail = NULL;
763     pctokenlist *cur;
764     pctoken *val;
765     symtype type;
766     AST *astcur;
767 
768     printf("********ENTERING constant_definition********\n
    ");
769 
770     EXPECT(idsym);
771     tokens.token = lasttoken;
772     tail = &tokens;
773 
774     while (pcp_accept(commasym)) {
775         /* update the linked list */
776         EXPECT(idsym);
777         ADDTOKEN(tail, lasttoken);
778     }
779 
780     EXPECT(eqsym);
781 
782     if (!(val = pcp_const_no_id(&type))) return 0;
783     NEXTTOKEN();
784 
785     /* add all the id's to the symbol table */
786     cur = &tokens;
787     while (cur != NULL) {
788         /* add to the symbol table */
789         if (!pcaddsym(cur->token->val.id, type, val->val,
    1, cur->token->lineno)) return 0;
790 
791         /* add to the parse tree */
792         astcur = AST_initialize(idasm);
793         astcur->name = strdup(cur->token->val.id);
794         AST_addchild(ast, astcur);
795 
796         /* free and go to the next */
797         tail = cur;
798         cur = cur->next;
799         if (tail != &tokens) free(tail);
800     }
801 
```

```
802     EXPECT(semicolonsym);
803     if (token->sym == idsym) return
    pcp_constant_definition(ast);
804
805     return 1;
806 }
807
808 int pcp_constant_definition_part(AST *ast) {
809     AST *astconst;
810
811     printf("********ENTERING constant_definition_part
    ********\n");
812
813     if (!pcp_accept(constsym)) return 1;
814
815     astconst = AST_initialize(constasm);
816     AST_addchild(ast, astconst);
817
818     return pcp_constant_definition(astconst);
819 }
820
821 int pcp_block(AST *ast) {
822     if (!pcp_constant_definition_part(ast)) return 0;
823     /*if (!pcp_type_definition_part()) return 0;*/
824     if (!pcp_variable_definition_part(ast)) return 0;
825     if (!pcp_procedure_and_function_definition_part(ast))
    return 0;
826     if (!pcp_statement_part(ast)) return 0;
827
828     return 1;
829 }
830
831 int pcp_program() {
832     EXPECT(programsym);
833
834     /* add to our symbol table */
835     EXPECT(idsym);
836     if (!pcaddsym(lasttoken->val.id, programtype, (symval)
    0, 0, lasttoken->lineno)) return 0;
837
838     /* add to our tree */
839     astroot = AST_initialize(programasm);
840     astroot->name = strdup(lasttoken->val.id);
841
842     EXPECT(semicolonsym);
843
844     if (!pcp_block(astroot)) return 0;
845
846     /*NEXTTOKEN();*/
847     if (token->sym != dotsym) {
```

```
848         return pcerror("[%u] pcp_expect: Unexpected symbol
    : %s vs %s\n", token->lineno, token->val, pcsymstr[dotsym
    ]);
849     }
850 
851     if (pcgettoken(fp) != NULL) {
852         pcerror("Expected end-of-file, but there is still
    content.");
853         return 0;
854     }
855 
856     return 1;
857 }
858 
859 int pcp_start() {
860     return pcp_program();
861 }
862 
863 int pcparse(FILE *ifp) {
864     fp = ifp;
865     lasttoken = token = nexttoken = pcgettoken(fp);
866     NEXTTOKEN();
867 
868     return pcp_start();
869 }
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * This is the parser program that recursively parses the
   scanned input file.
 5  */
 6 #ifndef PARSER_H
 7 #define PARSER_H
 8
 9 #include "scanner.h"
10
11 /* Parsers our input file */
12 int pcparse(FILE *fp);
13
14 #endif /* PARSER_H */
```

```
 1 %{
 2 #include "compiler.h"
 3 #include <stdio.h>
 4
 5 extern int yylex(void);
 6
 7 void yyerror (const char *s) {
 8     fprintf(stderr, "%s\n", s);
 9 }
10 %}
11
12 /* our lval types */
13 %union {
14     int ival;
15     double rval;
16     char *id;
17     char *string;
18     char chval;
19 }
20
21 /* our tokens */
22 %start program
23 %token LPAREN RPAREN LBRACK RBRACK /* ( | ) | [ | ] */
24 %token DOT COMMA SEMICOLON COLON /* . | , | ; | : */
25 %token ASSIGNOP LT GT LTE GTE NEQ EQ
    /* := | < | > | <= | >= | <> | = */
26 %token PROGRAM PROCEDURE FUNCTION /* program | procedure |
   function */
27 %token BEGINS END /* begin | end */
28 %token DO WHILE /* do | while */
29 %token IF THEN ELSE /* if | then | else */
30 %token AND OR NOT /* AND | OR | NOT */
31 %token VAR ARRAY /* var | ARRAY */
32 %token READ READLN WRITE WRITELN /* read | readln | write
    | writeln */
33 %token <chval> ADDOP MULOP /* + - | * / m d */
34 %token <ival> INTEGER INTNO /* integer */
35 %token <rval> REAL REALNO /* real */
36 %token <id> ID /* id */
37
38 %%
39
40 program:
41 PROGRAM ID
42 ;
43
44 %%
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the program that creates the symbol table in
 * linked list style from the parsed input
 */
#include "symtab.h"
#include "io.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

symtab *root    = NULL; /* our root table (keywords only) */
symtab *current = NULL; /* our current table */

const char *symtypestr[numsymtypes] = {
    /* Keywords */
    "keyword",

    /* Variable types */
    "char",
    "string",
    "integer",
    "real",

    /* Block types */
    "program",
    "procedure",
    "function",
    "block",

    /* nothingness */
    "notype"
};

/*
Lookup a lexeme with the option of restricting to the current scope.

@param name the name of the lexeme
@param current_scope_only whether to restrict to current scope
@return the entry or NULL if not found
*/
symentry *pclookupsym_internal(const char *name, int current_scope_only) {
    symentry    *entry;
    symtab      *tab;

```

```
47      if (!current) return NULL;
48
49      /* go through the current scope looking for the lexeme
    */
50      entry = current->entries;
51      while (entry) {
52          if (strcmp(entry->name, name) == 0) {
53              return entry;
54          }
55
56          entry = entry->next;
57      }
58
59      /* go up to the parent, if needed */
60      if (current_scope_only) return NULL;
61      tab = current->parent;
62      while (tab) {
63          entry = tab->entries;
64          while (entry) {
65              if (strcmp(entry->name, name) == 0) {
66                  return entry;
67              }
68
69              entry = entry->next;
70          }
71
72          /* keep going up */
73          tab = tab->parent;
74      }
75
76      /* never found */
77      return NULL;
78  }
79
80  int pcintializesymtab() {
81      symval val;
82
83      /* create or root table and set it current */
84      if (!(root = malloc(sizeof(*root)))) return 0;
85
86      current = root;
87      current->parent     = NULL;
88      current->entries    = NULL;
89      current->block      = NULL;
90
91      val.ival = 0;
92
93      /* this is expanded from scanner.c -> pcgetkeyword() */
94      /* TODO: make a pckeywords struct array */
95
```

```
 96     pcaddsym("div", keywordtype, (symval)"div", 1, 0);
 97     pcaddsym("mod", keywordtype, (symval)"mod", 1, 0);
 98
 99     pcaddsym("program", keywordtype, (symval)"program", 1
    , 0);
100     pcaddsym("procedure", keywordtype, (symval)"procedure"
    , 1, 0);
101     pcaddsym("function", keywordtype, (symval)"function",
    1, 0);
102     pcaddsym("begin", keywordtype, (symval)"begin", 1, 0);
103     pcaddsym("end", keywordtype, (symval)"end", 1, 0);
104
105     pcaddsym("and", keywordtype, (symval)"and", 1, 0);
106     pcaddsym("or", keywordtype, (symval)"or", 1, 0);
107     pcaddsym("not", keywordtype, (symval)"not", 1, 0);
108
109     pcaddsym("if", keywordtype, (symval)"if", 1, 0);
110     pcaddsym("else", keywordtype, (symval)"else", 1, 0);
111     pcaddsym("then", keywordtype, (symval)"then", 1, 0);
112     pcaddsym("do", keywordtype, (symval)"do", 1, 0);
113     pcaddsym("while", keywordtype, (symval)"while", 1, 0);
114
115     pcaddsym("array", keywordtype, (symval)"array", 1, 0);
116     pcaddsym("of", keywordtype, (symval)"of", 1, 0);
117     pcaddsym("char", keywordtype, (symval)"char", 1, 0);
118     pcaddsym("string", keywordtype, (symval)"string", 1, 0
    );
119     pcaddsym("integer", keywordtype, (symval)"integer", 1
    , 0);
120     pcaddsym("real", keywordtype, (symval)"real", 1, 0);
121     pcaddsym("var", keywordtype, (symval)"var", 1, 0);
122     pcaddsym("const", keywordtype, (symval)"const", 1, 0);
123
124     pcaddsym("chr", keywordtype, (symval)"chr", 1, 0);
125     pcaddsym("ord", keywordtype, (symval)"ord", 1, 0);
126     pcaddsym("read", keywordtype, (symval)"read", 1, 0);
127     pcaddsym("readln", keywordtype, (symval)"readln", 1, 0
    );
128     pcaddsym("write", keywordtype, (symval)"write", 1, 0);
129     pcaddsym("writeln", keywordtype, (symval)"writeln", 1
    , 0);
130
131     return 1;
132 }
133
134 void pcprintsymtabnode(symtab *node, unsigned depth) {
135     symentry *entry;
136     char tabs[21], *c;
137     int i;
138
```

```
139     if (!node) return;
140 
141     /* setup our tabs */
142     c = tabs;
143     for (i = 0; (i < depth && i < 20); ++i) {
144         *c++ = '\t';
145     }
146     *c = '\0';
147 
148     /* print self first */
149     entry = node->entries;
150     while (entry) {
151         switch (entry->type) {
152             case integertype:
153                 printf("%s%s (%s) : %d : %d\n", tabs,
    entry->name, symtypestr[entry->type], entry->lineno, entry
    ->val.ival);
154                 break;
155             case realtype:
156                 printf("%s%s (%s) : %d : %f\n", tabs,
    entry->name, symtypestr[entry->type], entry->lineno, entry
    ->val.rval);
157                 break;
158             case chartype:
159                 printf("%s%s (%s) : %d : %c\n", tabs,
    entry->name, symtypestr[entry->type], entry->lineno, entry
    ->val.cval);
160                 break;
161             case stringtype:
162                 printf("%s%s (%s) : %d : %s\n", tabs,
    entry->name, symtypestr[entry->type], entry->lineno, entry
    ->val.str);
163                 break;
164             default:
165                 printf("%s%s (%s) : %d : NULL\n", tabs,
    entry->name, symtypestr[entry->type], entry->lineno);
166         }
167 
168         /* print the symbol table for the child, if it
    exists */
169         if (entry->tab) pcprintsymtabnode(entry->tab,
    depth+1);
170 
171         /* go to the next entry */
172         entry = entry->next;
173     }
174 }
175 
176 void pcprintsymtab() {
177     printf("\n===== SYMBOL TABLE =====\n");
```

```
178     pcprintsymtabnode(root, 0);
179 }
180 
181 void pccleanupsymtabnode(symtab **tab) {
182     symentry *cur;
183 
184     cur = (*tab)->entries;
185     while (cur) {
186         /* cleanup children fist */
187         if (cur->tab) {
188             pccleanupsymtabnode(&(cur->tab));
189             cur->tab = NULL;
190         }
191 
192         /* cleanup name */
193         free((void*)(cur->name)); cur->name = NULL;
194 
195         /* next */
196         cur = cur->next;
197     }
198 
199     /* destroy our symtab */
200     free(*tab);
201     (*tab) = NULL;
202 }
203 
204 void pccleanupsymtab() {
205     pccleanupsymtabnode(&root);
206 }
207 
208 symentry *pcaddsym(const char *name, symtype type, symval
    val, int bconst, unsigned lineno) {
209     symentry *entry;
210 
211     /* make sure we have a root */
212     if (!current) {
213         pcerror("{%d} ERR: No symbol table defined.\n");
214         return NULL;
215     }
216 
217     /* make sure it doesn't yet exist in this scope */
218     if (pclookupsym_internal(name, 1)) {
219         pcerror("{%d} ERR: %s already exists in symbol
    table.\n", lineno, name);
220         return NULL;
221     }
222 
223     /* populate our entry */
224     if (!(entry = malloc(sizeof(*entry)))) return 0;
225     entry->name         = strdup(name);
```

```
226     entry->type         = type;
227     entry->val          = val;
228     entry->bconst       = bconst;
229     entry->lineno       = lineno;
230     entry->tab          = NULL;
231     entry->params       = NULL;
232     entry->returntype   = notype;
233
234     /* add to the tail of the entries */
235     entry->next         = current->entries;
236     current->entries    = entry;
237     return entry;
238 }
239
240 symentry *pcaddparam(const char *name, symtype type,
    unsigned lineno) {
241     symentry *entry;
242     symentry *func;
243     symparam *param;
244     symparam *cur;
245
246     /* make sure we're in a function/procedure */
247     if (!(func = current->block) || (func->type !=
    proceduretype && func->type != functiontype)) {
248         pcerror("{%d} ERR: Unable to determine function/
    procedure.\n", lineno);
249         return NULL;
250     }
251
252     /* add this to the symbol table */
253     if (!(entry = pcaddsym(name, type, (symval)0, 0,
    lineno))) return NULL;
254
255     /* update the params with the new param */
256     param = malloc(sizeof(*param));
257     param->entry = entry;
258     param->next = NULL;
259
260     /* add to the root if there isn't one here yet */
261     if (!(func->params)) {
262         func->params = param;
263         return entry;
264     }
265
266     /* add to the tail */
267     cur = func->params;
268     while (cur->next) cur = cur->next;
269     cur->next = param;
270
271     return entry;
```

```
272 }
273
274
275 symentry *pclookupsym(const char *name) {
276     return pclookupsym_internal(name, 0);
277 }
278
279 symentry *pcenterscope(const char *name, symtype type,
    unsigned lineno) {
280     symentry *entry;
281
282     if (!(entry = pcaddsym(name, type, (symval)0, 0,
    lineno))) return NULL;
283
284     /* create our table and make it the current, while
    updating it's parent */
285     if (!(entry->tab = malloc(sizeof(*(entry->tab)))))
    return NULL;
286     entry->tab->parent  = current;
287     entry->tab->entries = NULL;
288     entry->tab->block = entry;
289     current = entry->tab;
290
291     return entry;
292 }
293
294 int pcleavescope() {
295     /* can't leave if we're top dog */
296     if (current == root) return 0;
297
298     /* go up to our parent */
299     current = current->parent;
300     return 1;
301 }
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the program that creates the symbol table in
 * linked list style from the parsed input
 */

#ifndef SYMTAB_H
#define SYMTAB_H

#include "tokens.h"

/*
Symtype holds information about the type of an entry in the
 symbol table.
*/
typedef enum symtype {
    /* Keywords */
    keywordtype = 0,

    /* Variable types */
    chartype,
    stringtype,
    integertype,
    realtype,

    /* Block types */
    programtype,
    proceduretype,
    functiontype,
    blocktype,

    /* total count of types */
    notype,
    numsymtypes
} symtype;

/* Parameters for functions and procedures. */
struct symentry;
typedef struct symparam {
    struct symentry *entry;
    struct symparam *next;
} symparam;

/*
Array of string representation for each symtype.
KEEP UP TO DATE WITH symtype enum.
*/
extern const char *symtypestr[numsymtypes];

```

```
50 /*
51 Symentry holds links for our table.
52 */
53 struct symtab;
54 typedef struct symentry {
55     const char  *name;  /* name of the lexeme */
56     symtype     type;   /* type of the lexeme */
57     symval      val;    /* value of the lexeme */
58     unsigned    lineno; /* line the lexeme was declared on
    */
59     int         bconst; /* whether or not it's constant */
60
61     struct symtab   *tab;       /* symbol table for this
   entry (procedures and functions) */
62     symtype         returntype; /* return type for
   functions */
63     struct symparam *params;    /* paramaters */
64
65     /* link to the next entry */
66     struct symentry *next;
67 } symentry;
68
69 struct symtab {
70     struct symtab   *parent;    /* parent symtab */
71     symentry        *block;     /* the block entry that
   starts this */
72     symentry        *entries;   /* linked list of entries
    */
73 };
74
75 typedef struct symtab symtab;
76
77 /*
78 Initializes the symbol table with keywords.
79
80 @return 1 on success; 0 otherwise
81 */
82 int pcintializesymtab();
83
84 /*
85 Prints the symbol table.
86 */
87 void pcprintsymtab();
88
89 /*
90 Cleans up the symbol table.
91 */
92 void pccleanupsymtab();
93
94 /*
```

```
 95 Adds a value to the symbol table.
 96
 97 @param name the name of the lexeme
 98 @param type the type of the lexeme
 99 @param val the value of the lexeme
100 @param bconst 1 if constant, 0 otherwise
101 @param lineno the line the lexeme is declared on
102 @return 1 on success; 0 otherwise
103 */
104 symentry *pcaddsym(const char *name, symtype type, symval
    val, int bconst, unsigned lineno);
105
106 /*
107 Adds a variable to the symbol table as a parameter.
108
109 @param name the name of the lexeme
110 @param type the type of the lexeme
111 @param lineno the line the lexeme is declared on
112 @return 1 on success; 0 otherwise
113 */
114 symentry *pcaddparam(const char *name, symtype type,
    unsigned lineno);
115
116 /*
117 Lookup a symbol from the current table.
118
119 @param name the name of the lexeme
120 @return the entry or NULL if not found
121 */
122 symentry *pclookupsym(const char *name);
123
124 /*
125 Enters a new scope (creating a new symbol table and entry
    into
126 the current symbol table.
127
128 @param name the name of the lexeme
129 @param type the type of the lexeme
130 @param lineno the line the lexeme is declared on
131 @return 1 on success; 0 otherwise
132 */
133 symentry *pcenterscope(const char *name, symtype type,
    unsigned lineno);
134
135 /*
136 Leaves the current scope, returning to the parent scope.
137
138 @return 1 on success; 0 otherwise
139 */
140 int pcleavescope();
```

```
141
142
143 #endif /* SYMTAB_H */
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the program that defines the recognized tokens
 * for the program
 */
#include "tokens.h"
#include <stdlib.h>

const char *pcsymstr[numsyms] = {
    /* End-of-Tokens */
    "oefsym",

    /* Operators */
    "idivsym",
    "modsym",
    "addsym",
    "minussym",
    "multsym",
    "divsym",

    /* Scopes */
    "programsym",
    "proceduresym",
    "functionsym",
    "beginsym",
    "endsym",

    /* Boolean operators */
    "andsym",
    "orsym",
    "notsym",
    "ltsym",
    "ltesym",
    "neqsym",
    "gtsym",
    "gtesym",
    "eqsym",

    /* Punctuation */
    "assignsym",
    "colonsym",
    "semicolonsym",
    "commasym",
    "dotsym",
    "dotdotsym",
    "lparensym",
    "rparensym",
    "lbracksym",
    "rbracksym",
```

```
51
52      /* Control flow */
53      "ifsym",
54      "elsesym",
55      "thensym",
56      "dosym",
57      "whilesym",
58
59      /* Variables */
60      "idsym",
61      "arraysym",
62      "ofsym",
63      "charsym",
64      "stringsym",
65      "integersym",
66      "realsym",
67      "varsym",
68
69      /* Constants */
70      "integernosym",
71      "realnosym",
72      "stringvalsym",
73      "charvalsym",
74      "constsym",
75
76      /* Built-in functions */
77      "chrsym",
78      "ordsym",
79      "readsym",
80      "readlnsym",
81      "writesym",
82      "writelnsym",
83  };
84
85  pctoken *
86  pcnewtoken(pcsym sym, symval val, unsigned lineno) {
87      pctoken *token;
88
89      if (!(token = malloc(sizeof(*token)))) return NULL;
90      token->sym = sym;
91      token->val = val;
92      token->lineno = lineno;
93
94      return token;
95  }
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * This is the program that defines the recognized tokens
 5  * for the program
 6  */
 7
 8 #ifndef TOKENS_H
 9 #define TOKENS_H
10
11 /*
12 All of our possible sym value types.
13 */
14 typedef union symval {
15     int ival;
16     double rval;
17     char cval;
18     char *id;
19     char *str;
20 } symval;
21
22 /*
23 All of our possible tokens.
24 KEEP UP TO DATE WITH pcsymstr.
25 */
26 typedef enum {
27     /* End-of-Tokens */
28     eofsym = 0,
29
30     /* Operators */
31     idivsym,
32     modsym,
33     addsym,
34     minussym,
35     multsym,
36     divsym,
37
38     /* Scopes */
39     programsym,
40     proceduresym,
41     functionsym,
42     beginsym,
43     endsym,
44
45     /* Boolean operators */
46     andsym,
47     orsym,
48     notsym,
49     ltsym,
50     ltesym,
```

```
 51      neqsym,
 52      gtsym,
 53      gtesym,
 54      eqsym,
 55
 56      /* Punctuation */
 57      assignsym,
 58      colonsym,
 59      semicolonsym,
 60      commasym,
 61      dotsym,
 62      dotdotsym,
 63      lparensym,
 64      rparensym,
 65      lbracksym,
 66      rbracksym,
 67
 68      /* Control flow */
 69      ifsym,
 70      elsesym,
 71      thensym,
 72      dosym,
 73      whilesym,
 74
 75      /* Variables */
 76      idsym,
 77      arraysym,
 78      ofsym,
 79      charsym,
 80      stringsym,
 81      integersym,
 82      realsym,
 83      varsym,
 84
 85      /* Constants */
 86      integernosym,
 87      realnosym,
 88      stringvalsym,
 89      charvalsym,
 90      constsym,
 91
 92      /* Built-in functions */
 93      chrsym,
 94      ordsym,
 95      readsym,
 96      readlnsym,
 97      writesym,
 98      writelnsym,
 99
100      /* Number of syms */
```

```
101     numsyms
102 } pcsym;
103 
104 /*
105 Array of string representation for each sym.
106 KEEP UP TO DATE WITH pcsym enum.
107 */
108 extern const char *pcsymstr[numsyms];
109 
110 /* Structure for each token generated by our scanner. */
111 typedef struct pctoken {
112     pcsym       sym;
113     symval      val;
114     unsigned    lineno;
115 } pctoken;
116 
117 /*
118 Creates a new token with the given values.
119 
120 @param sym the sym type
121 @param val the value
122 @param lineno the line number
123 @return a malloc'd token or NULL on error
124 */
125 pctoken * pcnewtoken(pcsym sym, symval val, unsigned
    lineno);
126 
127 #endif /* TOKENS_H */
```

```
[program name:goodtestprogram]
    [const]
        [id name:myname]
        [id name:age]
        [id name:yearsalive]
    [var]
        [id name:x]
        [id name:y]
        [id name:z]
        [id name:a]
        [id name:b]
        [id name:c]
        [id name:f]
    [procedure name:testprocedure]
        [param]
            [id name:c1]
            [id name:c2]
        [var]
            [id name:c]
        [statement]
            [assign name:c]
                [id name:c]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c1]
            [assign name:c1]
                [id name:c1]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c2]
            [assign name:c2]
                [id name:c2]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c]
    [function name:testfunction]
        [param]
            [id name:c1]
            [id name:c2]
        [var]
            [id name:c]
        [statement]
            [assign name:c]
                [id name:c]
```

```
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [id name:c1]
                            [add]
                            [term]
                                [factor]
                                    [id name:c2]
    [procedure name:testprocedure2]
        [param]
            [id name:c1]
            [id name:c2]
        [var]
            [id name:c]
        [statement]
            [assign name:c]
                [id name:c]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c1]
            [assign name:c1]
                [id name:c1]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c2]
            [assign name:c2]
                [id name:c2]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c]
    [statement]
        [write]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [read]
            [id name:x]
        [write]
            [expr]
                [simexpr]
                    [term]
```

```
                            [factor]
                                [val val:Y]
        [write]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [id name:x]
        [assign name:f]
            [id name:f]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [assign name:f]
            [id name:f]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [assign name:f]
            [id name:f]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [assign name:f]
            [id name:f]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [assign name:z]
            [id name:z]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val]
        [while]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [expr]
                                [simexpr]
```

```
                                        [term]
                                            [factor]
                                                [id name:z]
                                    [rel]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [id name:x]
            [statement]
                [write]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [id name:z]
                [write]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [val val:Y]
                [assign name:z]
                    [id name:z]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [id name:z]
                            [add]
                            [term]
                                [factor]
                                    [val val:Y]
        [write]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [id name:z]
        [write]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [ord]
                                [expr]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [val val:Y
    ]
```

```
200          [assign name:y]
201              [id name:y]
202              [expr]
203                  [simexpr]
204                      [term]
205                          [factor]
206                              [id name:z]
207                          [mult]
208                          [factor]
209                              [expr]
210                                  [simexpr]
211                                      [term]
212                                          [factor]
213                                              [id name:z]
214                                      [add]
215                                      [term]
216                                          [factor]
217                                              [ord]
218                                                  [expr]
219                                                      [
    simexpr]
220                                                          [
    term]
221
        [factor]
222
            [val val:Y]
223                                      [add]
224                                      [term]
225                                          [factor]
226                                              [val val:Y]
227                      [add]
228                      [term]
229                          [factor]
230                              [id name:x]
231                          [mult]
232                          [factor]
233                              [val val:Y]
234          [write]
235              [expr]
236                  [simexpr]
237                      [term]
238                          [factor]
239                              [id name:y]
240          [assign name:z]
241              [id name:z]
242              [expr]
243                  [simexpr]
244                      [term]
245                          [factor]
```

```
246                                 [funccall name:testfunction]
247                                     [expr]
248                                         [simexpr]
249                                             [term]
250                                                 [factor]
251                                                     [id name:x
    ]
252                                     [expr]
253                                         [simexpr]
254                                             [term]
255                                                 [factor]
256                                                     [id name:y
    ]
257                         [add]
258                         [term]
259                             [factor]
260                                 [id name:x]
261             [if]
262                 [expr]
263                     [simexpr]
264                         [term]
265                             [factor]
266                                 [expr]
267                                     [simexpr]
268                                         [term]
269                                             [factor]
270                                                 [id name:x]
271                                     [rel]
272                                     [simexpr]
273                                         [term]
274                                             [factor]
275                                                 [id name:y]
276                 [statement]
277                     [write]
278                         [expr]
279                             [simexpr]
280                                 [term]
281                                     [factor]
282                                         [val val:Y]
283                 [statement]
284                     [write]
285                         [expr]
286                             [simexpr]
287                                 [term]
288                                     [factor]
289                                         [val val:Y]
290             [if]
291                 [expr]
292                     [simexpr]
293                         [term]
```

```
                            [factor]
                                [expr]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [id name:z]
                                    [rel]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [id name:x]
            [statement]
                [write]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [val val:Y]
        [assign name:x]
            [id name:x]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [while]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [expr]
                                [simexpr]
                                    [term]
                                        [factor]
                                            [id name:x]
                                [rel]
                                [simexpr]
                                    [term]
                                        [factor]
                                            [val val:Y]
            [statement]
                [write]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [chr]
                                        [expr]
                                            [simexpr]
                                                [term]
```

```
344                                                                 [
    factor]
345                                                                     [
    id name:x]
346                     [write]
347                         [expr]
348                             [simexpr]
349                                 [term]
350                                     [factor]
351                                         [val val:Y]
352                     [assign name:x]
353                         [id name:x]
354                         [expr]
355                             [simexpr]
356                                 [term]
357                                     [factor]
358                                         [id name:x]
359                                 [add]
360                                 [term]
361                                     [factor]
362                                         [val val:Y]
363             [write]
364                 [expr]
365                     [simexpr]
366                         [term]
367                             [factor]
368                                 [chr]
369                                     [expr]
370                                         [simexpr]
371                                             [term]
372                                                 [factor]
373                                                     [id name:x
    ]
374
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the scanner program that creates the tokens and lexemes for the input program
 */
#include "scanner.h"
#include "compiler.h"
#include "io.h"
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LINE_BUFF       2048

/* macro to recursively call pcgettoken when needed */
#define PCGETTOKEN_RECURSE(N, FP)   \
    if (N == EOF) return NULL;  \
    pcungetc(N, FP);                \
    return pcgettoken(FP);

int pcscanerrors = 0;
int pcscanwarnings = 0;

char line[LINE_BUFF];
char *lineptr = line;
size_t linesize = 0;

/*
Updates the current and next characters from the FILE stream.

@param cur the current character (will be overwritten)
@param next the next character (will be overwritten)
@param fp the FILE pointer
*/
void pcgetnextc(char *cur, char *next, FILE *fp) {
    *cur = *next;
    *next = pcgetc(fp);

    /* add the current character to our line for printing */
    if (linesize < LINE_BUFF && *cur != EOF) {
        *lineptr++ = *cur;
        ++linesize;
    }
}

```

```
48 /*
49 Appends the next character to the buffer at the given
   location and
50 increments the buffer.
51
52 @param b the current location in the buffer (will be
   overwritten)
53 @param cur the current character (will be overwritten)
54 @param next the next character (will be overwritten)
55 @param fp the FILE pointer
56 */
57 void pcappendnext(char **b, char *cur, char *next, FILE *fp
   ) {
58     pcgetnextc(cur, next, fp);
59     **b = *cur;
60     (*b)++;
61 }
62
63 /*
64 Determines if the character is a terminating character (i.e
   . punctuation).
65
66 @param c the character to check
67 @return 1 if terminating; 0 otherwise
68 */
69 int pcistermintor(char c) {
70     return (
71         c==':' || c==';' || c==',' || c=='.' ||
72         c=='=' || c=='>' || c=='<' ||
73         c=='(' || c==')' || c=='[' || c==']' ||
74         c=='+' || c=='-' || c=='*' || c=='/'
75     );
76 }
77
78 /*
79 Determines if the character signifies that we are at the
   end of the token
80 (i.e. EOF, whitespace, terminator) or some other random
   character that isn't
81 a letter or a number.
82
83 @param c the character to check
84 @return 1 if we should return; 0 otherwise
85 */
86 int pcisendoftoken(char c) {
87     return (
88         c == EOF || isspace(c) || pcistermintor(c)/* ||
89         (!isalpha(c) && !isdigit(c))*/
90     );
91 }
```

```
 92
 93 /*
 94 Determines if the character is a random character (i.e.
    not a end of token,
 95 not alpha, and not a number).
 96 */
 97 int pcisrandom(char c) {
 98     return (!pcisendoftoken(c) && !isalpha(c) && !isdigit(
    c));
 99 }
100
101 void pcresetline() {
102     /* print the line */
103     *lineptr = '\0';
104     printf("[%d] %s\n", pclineno, line);
105
106     /* reset the lineptr and size */
107     lineptr = line;
108     linesize = 0;
109 }
110
111 /*
112 Skips whitespace.
113 */
114 void pcskipwhitespace(char *cur, char *next, FILE *fp) {
115     int dontprint = 0;
116     while (isspace(*cur)) {
117         /* see if we have a new line */
118         if (*cur == '\n') {
119             /* don't print multiple blank lines */
120             if (!dontprint) {
121                 pcresetline();
122                 dontprint = 1;
123             } else {
124                 lineptr = line;
125                 linesize = 0;
126                 *lineptr = '\0';
127             }
128
129             /* increment our line counter */
130             ++pclineno;
131         }
132
133         pcgetnextc(cur, next, fp);
134     }
135 }
136
137 /*
138 Pulls a keyword from the given buffer, if available.
139
```

```
140 @param b the buffer the check
141 @param sym the pcsym to be updated
142 @return 1 on success; 0 on failure (not a keyword)
143 */
144 int pcgetkeyword(char *b, pcsym *sym) {
145     if (strcmp("div", b) == 0) *sym = idivsym;
146     else if (strcmp("mod", b) == 0) *sym = modsym;
147 
148     else if (strcmp("program", b) == 0) *sym = programsym;
149     else if (strcmp("procedure", b) == 0) *sym =
    proceduresym;
150     else if (strcmp("function", b) == 0) *sym =
    functionsym;
151     else if (strcmp("begin", b) == 0) *sym = beginsym;
152     else if (strcmp("end", b) == 0) *sym = endsym;
153 
154     else if (strcmp("and", b) == 0) *sym = andsym;
155     else if (strcmp("or", b) == 0) *sym = orsym;
156     else if (strcmp("not", b) == 0) *sym = notsym;
157 
158     else if (strcmp("if", b) == 0) *sym = ifsym;
159     else if (strcmp("else", b) == 0) *sym = elsesym;
160     else if (strcmp("then", b) == 0) *sym = thensym;
161     else if (strcmp("do", b) == 0) *sym = dosym;
162     else if (strcmp("while", b) == 0) *sym = whilesym;
163 
164     else if (strcmp("array", b) == 0) *sym = arraysym;
165     else if (strcmp("of", b) == 0) *sym = ofsym;
166     else if (strcmp("char", b) == 0) *sym = charsym;
167     else if (strcmp("string", b) == 0) *sym = stringsym;
168     else if (strcmp("integer", b) == 0) *sym = integersym;
169     else if (strcmp("real", b) == 0) *sym = realsym;
170     else if (strcmp("var", b) == 0) *sym = varsym;
171     else if (strcmp("const", b) == 0) *sym = constsym;
172 
173     else if (strcmp("chr", b) == 0) *sym = chrsym;
174     else if (strcmp("ord", b) == 0) *sym = ordsym;
175     else if (strcmp("read", b) == 0) *sym = readsym;
176     else if (strcmp("readln", b) == 0) *sym = readlnsym;
177     else if (strcmp("write", b) == 0) *sym = writesym;
178     else if (strcmp("writeln", b) == 0) *sym = writelnsym;
179 
180     /* unknown keyword */
181     else return 0;
182 
183     /* found a keyword; sym has been updated */
184     return 1;
185 }
186 
187 pctoken * pcgettoken(FILE *fp) {
```

```
188     char    cur, next,      /* current and next characters
     in the FILE */
189             *b, buf[255];   /* buffer filled while
    grabbing characters */
190     symval  val;            /* value of the token */
191     pcsym   sym;            /* sym of the token */
192
193     /* skip whitespace */
194     next = pcgetc(fp);
195     pcgetnextc(&cur, &next, fp);
196     pcskipwhitespace(&cur, &next, fp);
197
198     /* end-of-file? */
199     if (cur == EOF) {
200         pcresetline();
201         return NULL;
202     }
203
204     /* initialize our variables */
205     b   = buf;
206     *b  = '\0';
207     sym = eofsym;
208     val.ival = 0;
209
210     /* skip over single-line comments */
211     if (cur == '/') {
212         if (next == '/') {
213             /* consume up to the end of line */
214             while (next != '\n' && next != EOF) {
215                 pcgetnextc(&cur, &next, fp);
216             }
217
218             /* put the \n token back and get the next
    token */
219             PCGETTOKEN_RECURSE(next, fp);
220         }
221     }
222
223     /* skip over multi-line comments */
224     if (cur == '(' || cur == '{') {
225         char end1, end2;
226
227         /* determine our ending 2-char sequence */
228         if (cur == '(' && next == '*') {
229             end1 = '*';
230             end2 = ')';
231         } else if (cur == '{') {
232             end1 = '}';
233             end2 = 0;
234         } else {
```

```
235             end1 = 0;
236             end2 = 0;
237         }
238 
239         /* only skip if we have an ending sequence */
240         if (end1) {
241             /* store our starting lineno, since it will
    likely change */
242             unsigned startinglineno = pclineno;
243 
244             while (1) {
245                 /* match the first part */
246                 if (cur == end1) {
247                     /* only 1 to match, so leave our next
     */
248                     if (!end2) {
249                         break;
250                     }
251 
252                     /* 2 to match, so grab the next value
     */
253                     if (end2 && next == end2) {
254                         pcgetnextc(&cur, &next, fp);
255                         break;
256                     }
257                 }
258 
259                 /* warn if we hit the end of file without
    terminating */
260                 if (cur == EOF) {
261                     if (end2) {
262                         pcerror("{%d} ERR: Multiline
    comment missing termintors: %c%c", startinglineno, end1,
    end2);
263                     } else {
264                         pcerror("{%d} ERR: Multiline
    comment missing termintor: %c", startinglineno, end1);
265                     }
266 
267                     return NULL;
268                 }
269 
270                 /* add to our linecount on \n */
271                 if (cur == '\n') {
272                     ++pclineno;
273                 }
274 
275                 /* keep skipping characters */
276                 pcgetnextc(&cur, &next, fp);
277             }
```

```
278
279             /* put the next token back and get the next
    token */
280             PCGETTOKEN_RECURSE(next, fp);
281         }
282     }
283
284     /* check the terminators */
285     if (pcistermintor(cur)) {
286         switch (cur) {
287             case '(':   sym = lparensym; break;
288             case ')':   sym = rparensym; break;
289             case '[':   sym = lbracksym; break;
290             case ']':   sym = rbracksym; break;
291             case ';':   sym = semicolonsym; break;
292             case ',':   sym = commasym; break;
293             case '.':
294                 if (next == '.') {
295                     sym = dotdotsym;
296                     pcgetnextc(&cur, &next, fp);
297                 } else {
298                     sym = dotsym;
299                 }
300                 break;
301             case ':':
302                 if (next == '=') {
303                     sym = assignsym;
304                     pcgetnextc(&cur, &next, fp);
305                 } else {
306                     sym = colonsym;
307                 }
308                 break;
309             case '=':   sym = eqsym; break;
310             case '<':
311                 if (next == '=') {
312                     sym = ltesym;
313                     pcgetnextc(&cur, &next, fp);
314                 } else if (next == '>') {
315                     sym = neqsym;
316                     pcgetnextc(&cur, &next, fp);
317                 } else {
318                     sym = ltsym;
319                 }
320                 break;
321             case '>':
322                 if (next == '=') {
323                     sym = gtesym;
324                     pcgetnextc(&cur, &next, fp);
325                 } else {
326                     sym = gtsym;
```

```
327                 }
328                 break;
329             case '+':   sym = addsym; break;
330             case '-':   sym = minussym; break;
331             case '*':   sym = multsym; break;
332             case '/':   sym = divsym; break;
333         }
334     }
335     /* now check for a number */
336     else if (isdigit(cur)) {
337         *b++ = cur;
338 
339         /* keep adding digits until the next isn't a digit
     */
340         while (isdigit(next)) {
341             pcappendnext(&b, &cur, &next, fp);
342         }
343 
344         /* see if we have a dot and shift to real digit */
345         if (next == '.') {
346             pcappendnext(&b, &cur, &next, fp);
347 
348             /* keep adding digits until the next isn't a
    digit */
349             while (isdigit(next)) {
350                 pcappendnext(&b, &cur, &next, fp);
351             }
352 
353             /* see if we have a e or E and shift to
    scientific */
354             if (next == 'e' || next == 'E') {
355                 pcappendnext(&b, &cur, &next, fp);
356 
357                 /* check for +/- */
358                 if (next == '+' || next == '-') {
359                     pcappendnext(&b, &cur, &next, fp);
360                 }
361 
362                 /* keep adding digits */
363                 while (isdigit(next)) {
364                     pcappendnext(&b, &cur, &next, fp);
365                 }
366 
367                 /* if we don't have a terminal/whitespace
    now, ill formed real number */
368                 if (!pcisendoftoken(next)) {
369                     /* keep consuming until we do hit a
    space or terminator */
370                     while (!pcisendoftoken(next)) {
371                         pcappendnext(&b, &cur, &next, fp);
```

```
372                     }
373
374                     /* print the error */
375                     *b = '\0';
376                     pcerror("{%d} ERR: Ill formed real
    number: %s\n", pclineno, buf);
377                     ++pcscanerrors;
378
379                     /* go to the next token */
380                     PCGETTOKEN_RECURSE(next, fp);
381                 }
382             }
383             /* if we don't have a terminal / whitespace /
    random char, ill formed real number */
384             else if (!pcisendoftoken(next)) {
385                     /* keep consuming until we do hit a
    space or terminator */
386                     while (!pcisendoftoken(next)) {
387                         pcappendnext(&b, &cur, &next, fp);
388                     }
389
390                     /* print the error */
391                     *b = '\0';
392                     pcerror("{%d} ERR: Ill formed real
    number: %s\n", pclineno, buf);
393                     ++pcscanerrors;
394
395                     /* go to the next token */
396                     PCGETTOKEN_RECURSE(next, fp);
397             }
398
399             /* we have a legitimate real number! calculate
     and create our token */
400             *b = '\0';
401             val.rval = atof(buf);
402             sym = realnosym;
403         }
404
405         /* if we don't have a end of token, ill formed
    integer number */
406         else if (!pcisendoftoken(next)) {
407             /* keep consuming until we do hit a space or
    terminator */
408             while (!pcisendoftoken(next)) {
409                 pcappendnext(&b, &cur, &next, fp);
410             }
411
412             /* print the error */
413             *b = '\0';
414             pcerror("{%d} ERR: Ill formed integer number
```

```
414 or id: %s\n", pclineno, buf);
415             ++pcscanerrors;
416 
417             /* go to the next token */
418             PCGETTOKEN_RECURSE(next, fp);
419         }
420 
421         /* we have a good integer! */
422         else {
423             *b = '\0';
424             val.ival = atoi(buf);
425             sym = integernosym;
426         }
427     }
428 
429     /* now check for strings and characters */
430     else if (cur == '\'') {
431         int scount = 0;
432 
433         /* add values to the buffer until we hit a \n or
     ' or EOF */
434         while (next != '\n' && next != '\'' && next != EOF
    ) {
435             pcgetnextc(&cur, &next, fp);
436             *b++ = cur;
437             ++scount;
438         }
439 
440         /* if we hit a new line or EOF, then we have an
    ill-formed string */
441         if (next == '\n' || next == EOF) {
442             /* print the error */
443             *b = '\0';
444             pcerror("{%d} ERR: No closing ': %s\n",
    pclineno, buf);
445             ++pcscanerrors;
446 
447             /* go to the next token */
448             PCGETTOKEN_RECURSE(next, fp);
449         }
450 
451         /* warn about empty strings */
452         *b = '\0';
453         if (!scount) {
454             pcerror("{%d} WARN: Empty string/character
    found.\n", pclineno);
455             ++pcscanwarnings;
456         }
457 
458         /* prepare our character if 1 value */
```

```
459         if (scount == 1) {
460             sym = charvalsym;
461             val.cval = *buf;
462         }
463         /* otherwise, it's a string */
464         else {
465             sym = stringvalsym;
466             val.str = strdup(buf);
467         }
468 
469         /* we consume another from the stream, so the tick
     doesn't go back in */
470         pcgetnextc(&cur, &next, fp);
471     }
472 
473     /* now check for keywords and id's */
474     else if (isalpha(cur)) {
475         *b++ = cur;
476 
477         /* consume letters and numbers */
478         while (isalpha(next) || isdigit(next)) {
479             pcappendnext(&b, &cur, &next, fp);
480         }
481 
482         /* make sure we have an end of token */
483         if (!pcisendoftoken(next)) {
484             while (!pcisendoftoken(next)) {
485                 pcappendnext(&b, &cur, &next, fp);
486             }
487 
488             /* print the error */
489             *b = '\0';
490             pcerror("{%d} ERR: Ill formed keyword or id: %
    s\n", pclineno, buf);
491             ++pcscanerrors;
492 
493             /* go to the next token */
494             PCGETTOKEN_RECURSE(next, fp);
495         }
496 
497         /* determine what kind of symbol we have */
498         *b = '\0';
499         strtolower(buf);
500         if (!pcgetkeyword(buf, &sym)) {
501             sym = idsym;
502             val.id = strdup(buf);
503         }
504     }
505 
506     /* unknown character */
```

```
507     else {
508         pcerror("{%d} ERR: Unknown character: %c\n",
    pclineno, cur);
509         ++pcscanerrors;
510 
511         /* get the next token */
512         PCGETTOKEN_RECURSE(next, fp);
513     }
514 
515     /* unget our next value (so it's our current in next
    call) */
516     if (next != EOF) pcungetc(next, fp);
517 
518     /* generate and return our token */
519     return pcnewtoken(sym, val, pclineno);
520 }
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * This is the scanner program that creates the tokens and lexemes for the input program
 */

#ifndef SCANNER_H
#define SCANNER_H

#include "tokens.h"
#include <stdio.h>

extern unsigned pclineno;
extern int pcscanerrors;
extern int pcscanwarnings;

/*
Gets the next token from the stream, or NULL if consumed.

@param fp the FILE pointer
@return a malloc'd next token, or NULL if consumed
*/
pctoken *pcgettoken(FILE *fp);

#endif /* SCANNER_H */
```

```
 1 /*
 2 scanner.l hold the regex information for creating the
   lexemes for our
 3 compiler.
 4 */
 5
 6 %{
 7 #include "compiler.h"
 8 #include "parser.tab.h"
 9
10 #include <math.h>
11 #include <string.h>
12 %}
13
14 %option caseless
15
16 %x MLCSTAR
17 %x MLCBRACE
18
19 addop       [+-]
20 mulop       [*/]|"mod"|"div"
21 digit       [0-9]
22 real        {digit}+\.{digit}+([eE][+-]?{digit}+)?
23 id          [a-z][a-z0-9]*
24 whitespace  [ \t\r]+
25
26 %%
27
28 \(              {DEBUG_PRINTF(("< LPAREN >\n")); return
   LPAREN; }
29 \)              {DEBUG_PRINTF(("< RPAREN >\n")); return
   RPAREN; }
30 \[              {DEBUG_PRINTF(("< LBRACK >\n")); return
   LBRACK; }
31 \]              {DEBUG_PRINTF(("< RBRACK >\n")); return
   RBRACK; }
32
33 \.              {DEBUG_PRINTF(("< DOT >\n")); return DOT; }
34 \,              {DEBUG_PRINTF(("< COMMA >\n")); return
   COMMA; }
35 \;              {DEBUG_PRINTF(("< SEMICOLON >\n")); return
   SEMICOLON; }
36 \:              {DEBUG_PRINTF(("< COLON >\n")); return
   COLON; }
37
38 {addop}         {yylval.chval = yytext[0]; DEBUG_PRINTF
   (("< ADDOP , %c >", yylval.chval)); return ADDOP; }
39 {mulop}         {yylval.chval = yytext[0]; DEBUG_PRINTF
   (("< MULOP , %c >", yylval.chval)); return MULOP; }
40
```

```
41 ":="              {DEBUG_PRINTF(("< ASSIGNOP >\n")); return
   ASSIGNOP; }
42 "<"               {DEBUG_PRINTF(("< LT >\n")); return LT; }
43 ">"               {DEBUG_PRINTF(("< GT >\n")); return GT; }
44 "<="              {DEBUG_PRINTF(("< LTE >\n")); return LTE; }
45 ">="              {DEBUG_PRINTF(("< GTE >\n")); return GTE; }
46 "<>"              {DEBUG_PRINTF(("< NEQ >\n")); return NEQ; }
47 "="               {DEBUG_PRINTF(("< EQ >\n")); return EQ; }
48
49 "program"         {DEBUG_PRINTF(("< PROGRAM >\n")); return
   PROGRAM; }
50 "procedure"       {DEBUG_PRINTF(("< PROCEDURE >\n")); return
   PROCEDURE; }
51 "function"        {DEBUG_PRINTF(("< FUNCTION >\n")); return
   FUNCTION; }
52
53 "begin"           {DEBUG_PRINTF(("< BEGINS >\n")); return
   BEGINS; }
54 "end"             {DEBUG_PRINTF(("< END >\n")); return END; }
55
56 "do"              {DEBUG_PRINTF(("< DO >\n")); return DO; }
57 "while"           {DEBUG_PRINTF(("< WHILE >\n")); return
   WHILE; }
58
59 "if"              {DEBUG_PRINTF(("< IF >\n")); return IF; }
60 "then"            {DEBUG_PRINTF(("< THEN >\n")); return THEN
   ; }
61 "else"            {DEBUG_PRINTF(("< ELSE >\n")); return ELSE
   ; }
62
63 "and"             {DEBUG_PRINTF(("< AND >\n")); return AND; }
64 "or"              {DEBUG_PRINTF(("< OR >\n")); return OR; }
65 "not"             {DEBUG_PRINTF(("< NOT >\n")); return NOT; }
66
67 "var"             {DEBUG_PRINTF(("< VAR >\n")); return VAR; }
68 "array"           {DEBUG_PRINTF(("< ARRAY >\n")); return
   ARRAY; }
69
70 "read"            {DEBUG_PRINTF(("< READ >\n")); return READ
   ; }
71 "readln"          {DEBUG_PRINTF(("< READLN >\n")); return
   READLN; }
72 "write"           {DEBUG_PRINTF(("< WRITE >\n")); return
   WRITE; }
73 "writeln"         {DEBUG_PRINTF(("< WRITELN >\n")); return
   WRITELN; }
74
75 "integer"         {DEBUG_PRINTF(("< INTEGER >\n")); return
   INTEGER; }
76 {digit}           {yylval.ival = atoi(yytext); DEBUG_PRINTF
```

```
 76 (("< INTNO , %d >\n", yylval.ival)); return INTNO; }
 77
 78 "real"          {DEBUG_PRINTF(("< REAL >\n")); return REAL
    ; }
 79 {real}          {yylval.rval = atof(yytext); DEBUG_PRINTF
    (("< REALNO, %f >\n", yylval.rval)); return REALNO; }
 80
 81 {id}            {yylval.id = strtolower(strdup(yytext));
    DEBUG_PRINTF(("< ID , %s >\n", yylval.id)); return ID; }
 82
 83 {whitespace}    { /* whitespace */ }
 84 \n              { DEBUG_PRINTF(("[%d]\n\n", yylineno)); ++
    yylineno; }
 85
 86 "//".*          { /* skip comment to end of line */ }
 87 "(*"            {BEGIN(MLCSTAR); }
 88 "{"             {BEGIN(MLCBRACE); }
 89
 90 .               {fprintf(stderr, "{%d} Unknown character
    : %s\n", yylineno, yytext); }
 91
 92 <MLCSTAR>"*)"       {BEGIN(INITIAL); }
 93 <MLCSTAR>[^*\n]+    { /* eat comment in chunks */ }
 94 <MLCSTAR>"*"        { /* eat the lone star */ }
 95 <MLCSTAR>\n         { yylineno++; }
 96
 97 <MLCBRACE>"}"       {BEGIN(INITIAL); }
 98 <MLCBRACE>[^\n]+    { /* eat comment in chunks */ }
 99 <MLCBRACE>\n        { yylineno++; }
100
101 %%
```

```
 1 ## EDITORS
 2 *~*
 3 *.bak
 4 *.swp
 5 *.tmp
 6 *.log
 7
 8 ## WINDOWS
 9 Thumbs.db
10 Desktop.ini
11
12 ## MAC
13 .DS_Store
14
15 ## C
16 *.o
17
18 ## Flex / Bison
19 *.output
20 lex.yy.c
21 *.tab.c
22 *.tab.h
23
24 ## Programs
25 mini-pascal-compiler
26 yy-mini-pascal-compiler
27
```

```
 1 /*
 2  * @author Derek Trom
 3  * @author Elena Corpus
 4  * This is the main program that drives everything
 5  * for the compiler
 6  */
 7 #include "compiler.h"
 8
 9 #ifndef YYCOMPILE
10 # include "tokens.h"
11 # include "scanner.h"
12 # include "parser.h"
13 # include "symtab.h"
14 # include "io.h"
15 # include "ast.h"
16 #endif /* YYCOMPILE */
17
18 #include <stdio.h>
19 #include <stdlib.h>
20 #include <ctype.h>
21
22 #ifdef YYCOMPILE
23     extern FILE *yyin;
24     extern int yylex(void);
25 #else
26     extern int pcscanerrors;
27 #endif /* YYCOMPILE */
28
29 void
30 usage(const char *progname) {
31     printf("Usage: %s filename\n  filename\tPascal file to
   compile\n", progname);
32 }
33
34 char *
35 strtolower(char *s) {
36     char *c = s;
37     for ( ; *c; ++c) {
38         *c = tolower(*c);
39     }
40
41     return s;
42 }
43
44 int
45 main(int argc, char **argv) {
46     FILE *fp;
47     char *filename;
48 #ifdef YYCOMPILE
49     int token;
```

```
50 #else
51     pctoken *token;
52     pctoken *nexttoken;
53 #endif /* YYCOMPILE */
54 
55     /* read the filename from command line */
56     if (argc > 1) {
57         filename = argv[1];
58 
59         /* open our file */
60         fp = fopen(filename, "r");
61         if (!fp) {
62             printf("Unable to open file: %s.\n", filename);
63             return EXIT_FAILURE;
64         }
65 
66         #ifdef YYCOMPILE
67             yyin = fp;
68         #endif
69     } else {
70         usage(argv[0]);
71         return EXIT_FAILURE;
72     }
73 
74     printf("Reading file: %s\n\n", filename);
75 
76     /** just run the lexer for now, skipping the scanner */
77 #ifdef YYCOMPILE
78     while ((token = yylex())) {
79         //printf("%d\n", token);
80     }
81 #else
82     /* initialize our symbol table */
83     pcintializesymtab();
84 
85     if (pcparse(fp)) {
86         printf("\nPARSING COMPLETED SUCCESSFULLY!!!!!\n");
87     } else {
88         printf("\nERRORS PARSING!!!!!\n");
89         pcscanerrors = 1;
90     }
91 
92     /* spit out errors */
93     if (pcscanerrors) {
94         printf("\n%d ERRORS during scanning!\n",
   pcscanerrors);
95     }
96 
97     /* save our AST tree */
98     FILE *astfp;
```

```
 99     if ((astfp = fopen("astfp.txt", "w"))) {
100         AST_print(astroot, astfp);
101         printf("\nSaved astfp.txt\n");
102     }
103     AST_cleanup(&astroot);
104 
105     /* print our symbol table */
106     pcprintsymtab();
107     pccleanupsymtab();
108 #endif /* YYCOMPILE */
109 
110     return EXIT_SUCCESS;
111 }
```

```c
/*
 * @author Derek Trom
 * @author Elena Corpus
 * compiler.h is the main entry point of the program and is responsible
 * for handling user input and running the other portions of the compiler,
 * such as the scanner and parser.
*/

#ifndef COMPILER_H
#define COMPILER_H

/* MACRO for debug printing */
#ifdef DEBUG
# define DEBUG_PRINTF(x) printf x
#else
# define DEBUG_PRINTF(x) do {} while (0)
#endif

/*
Transforms a string to a lower-case alternative.
Assumes that the string is NUL-terminated.

@param s the string to turn to lowercase
@return pointer to the front of s
*/
char *strtolower(char *s);

#endif /* COMPILER_H */
```

```
 1 ## default LF normalization
 2 * text=auto
 3
 4 ## standard msysgit
 5 *.doc   diff=astextplain
 6 *.DOC   diff=astextplain
 7 *.docx  diff=astextplain
 8 *.DOCX  diff=astextplain
 9 *.dot   diff=astextplain
10 *.DOT   diff=astextplain
11 *.mpp   diff=astextplain
12 *.MPP   diff=astextplain
13 *.pdf   diff=astextplain
14 *.PDF   diff=astextplain
15 *.rtf   diff=astextplain
16 *.RTF   diff=astextplain
17 *.vsdx  diff=astextplain
18 *.VSDX  diff=astextplain
```

```
1 webprint/
2
```

```
[program name:goodtestprogram]
    [const]
        [id name:myname]
        [id name:age]
        [id name:yearsalive]
    [var]
        [id name:x]
        [id name:y]
        [id name:z]
        [id name:a]
        [id name:b]
        [id name:c]
        [id name:f]
    [procedure name:testprocedure]
        [param]
            [id name:c1]
            [id name:c2]
        [var]
            [id name:c]
        [statement]
            [assign name:c]
                [id name:c]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c1]
            [assign name:c1]
                [id name:c1]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c2]
            [assign name:c2]
                [id name:c2]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c]
    [function name:testfunction]
        [param]
            [id name:c1]
            [id name:c2]
        [var]
            [id name:c]
        [statement]
            [assign name:c]
                [id name:c]
```

```
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [id name:c1]
                            [add]
                            [term]
                                [factor]
                                    [id name:c2]
    [procedure name:testprocedure2]
        [param]
            [id name:c1]
            [id name:c2]
        [var]
            [id name:c]
        [statement]
            [assign name:c]
                [id name:c]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c1]
            [assign name:c1]
                [id name:c1]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c2]
            [assign name:c2]
                [id name:c2]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [id name:c]
    [statement]
        [write]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [read]
            [id name:x]
        [write]
            [expr]
                [simexpr]
                    [term]
```

```
                            [factor]
                                [val val:Y]
        [write]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [id name:x]
        [assign name:f]
            [id name:f]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [assign name:f]
            [id name:f]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [assign name:f]
            [id name:f]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [assign name:f]
            [id name:f]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [assign name:z]
            [id name:z]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val]
        [while]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [expr]
                                [simexpr]
```

```
                                        [term]
                                            [factor]
                                                [id name:z]
                                    [rel]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [id name:x]
            [statement]
                [write]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [id name:z]
                [write]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [val val:Y]
                [assign name:z]
                    [id name:z]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [id name:z]
                            [add]
                            [term]
                                [factor]
                                    [val val:Y]
        [write]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [id name:z]
        [write]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [ord]
                                [expr]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [val val:Y
    ]
```

```
200          [assign name:y]
201              [id name:y]
202              [expr]
203                  [simexpr]
204                      [term]
205                          [factor]
206                              [id name:z]
207                          [mult]
208                          [factor]
209                              [expr]
210                                  [simexpr]
211                                      [term]
212                                          [factor]
213                                              [id name:z]
214                                      [add]
215                                      [term]
216                                          [factor]
217                                              [ord]
218                                                  [expr]
219                                                      [
    simexpr]
220                                                          [
    term]
221
        [factor]
222
            [val val:Y]
223                                      [add]
224                                      [term]
225                                          [factor]
226                                              [val val:Y]
227                      [add]
228                      [term]
229                          [factor]
230                              [id name:x]
231                          [mult]
232                          [factor]
233                              [val val:Y]
234          [write]
235              [expr]
236                  [simexpr]
237                      [term]
238                          [factor]
239                              [id name:y]
240          [assign name:z]
241              [id name:z]
242              [expr]
243                  [simexpr]
244                      [term]
245                          [factor]
```

```
246                                 [funccall name:testfunction]
247                                     [expr]
248                                         [simexpr]
249                                             [term]
250                                                 [factor]
251                                                     [id name:x
    ]
252                                     [expr]
253                                         [simexpr]
254                                             [term]
255                                                 [factor]
256                                                     [id name:y
    ]
257                         [add]
258                         [term]
259                             [factor]
260                                 [id name:x]
261             [if]
262                 [expr]
263                     [simexpr]
264                         [term]
265                             [factor]
266                                 [expr]
267                                     [simexpr]
268                                         [term]
269                                             [factor]
270                                                 [id name:x]
271                                     [rel]
272                                     [simexpr]
273                                         [term]
274                                             [factor]
275                                                 [id name:y]
276                 [statement]
277                     [write]
278                         [expr]
279                             [simexpr]
280                                 [term]
281                                     [factor]
282                                         [val val:Y]
283                 [statement]
284                     [write]
285                         [expr]
286                             [simexpr]
287                                 [term]
288                                     [factor]
289                                         [val val:Y]
290             [if]
291                 [expr]
292                     [simexpr]
293                         [term]
```

```
                            [factor]
                                [expr]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [id name:z]
                                    [rel]
                                    [simexpr]
                                        [term]
                                            [factor]
                                                [id name:x]
            [statement]
                [write]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [val val:Y]
        [assign name:x]
            [id name:x]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [val val:Y]
        [while]
            [expr]
                [simexpr]
                    [term]
                        [factor]
                            [expr]
                                [simexpr]
                                    [term]
                                        [factor]
                                            [id name:x]
                                [rel]
                                [simexpr]
                                    [term]
                                        [factor]
                                            [val val:Y]
            [statement]
                [write]
                    [expr]
                        [simexpr]
                            [term]
                                [factor]
                                    [chr]
                                        [expr]
                                            [simexpr]
                                                [term]
```

```
                                                                [
    factor]
                                                                    [
    id name:x]
                    [write]
                        [expr]
                            [simexpr]
                                [term]
                                    [factor]
                                        [val val:Y]
                    [assign name:x]
                        [id name:x]
                        [expr]
                            [simexpr]
                                [term]
                                    [factor]
                                        [id name:x]
                                [add]
                                [term]
                                    [factor]
                                        [val val:Y]
            [write]
                [expr]
                    [simexpr]
                        [term]
                            [factor]
                                [chr]
                                    [expr]
                                        [simexpr]
                                            [term]
                                                [factor]
                                                    [id name:x
    ]
```

```
1 *.txt
2
```

```
 1 program BadTestProgram;
 2   var
 3     x, y, z : integer
 4     a, b, c : char
 5     f : float
 6   ;
 7 begin
 8   write('Enter a number to count to from 0: ');
 9   read(x);
10   write('You entered: ');
11   writeln(x);
12
13   f := 3.25e-15;
14   f := 16.94x;  // Real error
15
16   x := 158j;  // Integer error
17   1x := 6;  // id error
18
19   z := 0;
20   while (z < x) do
21     begin
22       write(z); write(', ');
23       z := z + 1
24     end;
25   writeln(z);
26
27   x := b; // Type error
28   x := z % 5; // Unknown character
29
30   writeln(ord('0'));
31   y := z * (z + ord('0') - 3) + x div 2;
32   writeln(y);
33
34   if (x > y) then
35     begin
36       writeln('x is bigger than y!');
37     end
38   else
39     begin
40       writeln('x is smaller than y!');
41     end;
42
43   if (z = x) then
44     begin
45       writeln('z is equal to x');
46     end;
47
48   x := 65
49
50   while (x < 90) do
```

```
51     begin
52       write(chr(x)); write(', ');
53       x := x + 1;
54     end;
55   writeln('One more to go!) // Quote error
56   writeln(chr(x));
57 end.
```

```
 1 derektrom@Raelyns-MBP mini-pascal-compiler-V2 % ./mini-
   pascal-compiler tests/badTest.pas
 2 Reading file: tests/badTest.pas
 3
 4 < programsym >
 5 < idsym , badtestprogram >
 6 [1] program BadTestProgram;
 7
 8 < semicolonsym >
 9 [2]     var
10
11 < varsym >
12 ********ENTERING constant_definition_part********
13 ********ENTERING variable_definition********
14 < idsym , x >
15 ********ENTERING variable_definition********
16 < commasym >
17 < idsym , y >
18 < commasym >
19 < idsym , z >
20 < colonsym >
21 [3]         x, y, z : integer
22
23 < integersym >
24 < idsym , a >
25 [4] pcp_expect: Unexpected symbol: a vs semicolonsym
26
27 ERRORS PARSING!!!!!
28
29 1 ERRORS during scanning!
30
31 Saved astfp.txt
32
33 ===== SYMBOL TABLE =====
34 z (integer) : 3 : 0
35 y (integer) : 3 : 0
36 x (integer) : 3 : 0
37 badtestprogram (program) : 1 : NULL
38 writeln (keyword) : 0 : NULL
39 write (keyword) : 0 : NULL
40 readln (keyword) : 0 : NULL
41 read (keyword) : 0 : NULL
42 ord (keyword) : 0 : NULL
43 chr (keyword) : 0 : NULL
44 const (keyword) : 0 : NULL
45 var (keyword) : 0 : NULL
46 real (keyword) : 0 : NULL
47 integer (keyword) : 0 : NULL
48 string (keyword) : 0 : NULL
49 char (keyword) : 0 : NULL
```

```
50 of (keyword) : 0 : NULL
51 array (keyword) : 0 : NULL
52 while (keyword) : 0 : NULL
53 do (keyword) : 0 : NULL
54 then (keyword) : 0 : NULL
55 else (keyword) : 0 : NULL
56 if (keyword) : 0 : NULL
57 not (keyword) : 0 : NULL
58 or (keyword) : 0 : NULL
59 and (keyword) : 0 : NULL
60 end (keyword) : 0 : NULL
61 begin (keyword) : 0 : NULL
62 function (keyword) : 0 : NULL
63 procedure (keyword) : 0 : NULL
64 program (keyword) : 0 : NULL
65 mod (keyword) : 0 : NULL
66 div (keyword) : 0 : NULL
67 derektrom@Raelyns-MBP mini-pascal-compiler-V2 %
68
```

```
 1 program GoodTestProgram;
 2   const
 3     myname = 'Derek';
 4     age, yearsalive = 27;
 5   var
 6     x, y, z : integer;
 7     a, b, c : char;
 8     f : real;
 9
10   procedure testProcedure (c1,c2:integer);
11     var
12       c : integer;
13     begin
14       c := c1;
15       c1 := c2;
16       c2 := c;
17     end;
18
19   function testFunction (c1,c2:integer) : integer;
20     var
21       c : integer;
22     begin
23       c := c1+c2;
24     end;
25
26   procedure testProcedure2 (c1,c2:integer);
27     var
28       c : integer;
29     begin
30       c := c1;
31       c1 := c2;
32       c2 := c;
33     end;
34 begin
35   // This line will be skipped
36   write('Enter a number: ');
37   read(x);
38   write('You entered: ');
39   writeln(x);
40
41   // Test some real numbers
42   f := 15.3;
43   f := 3.25e-15;
44   f := 19.27e+8;
45   f := 55.2e10;
46
47   z := 0;
48   while (z < x) do
49     begin
50       write(z); write(', ');
```

```
51        z := z + 1;
52      end;
53    writeln(z);
54
55    writeln(ord('0')); // Rest of this line skipped
56    y := z * (z + (* inline comment *) ord('0') - 3) + x div
    2;
57    writeln(y);
58
59    z := testFunction(x, y) + x;
60
61    { Muliline
62    comment
63    }
64    if (x > y) then
65      begin
66        writeln('x is bigger than y!');
67      end
68    else
69      begin
70        writeln('x is smaller than y!');
71      end;
72
73    (* Another
74    multline
75    comment
76    *)
77    if (z = x) then
78      begin
79        writeln('z is equal to x');
80      end;
81
82    x := 65;
83
84    while (x < 90) do
85      begin
86        write({ another inline comment}chr(x)); write(', ');
87        x := x + 1;
88      end;
89    writeln(chr(x));
90  end.
```

```
derektrom@Raelyns-MBP mini-pascal-compiler-V2 % ./mini-pascal-compiler tests/goodTest.pas
Reading file: tests/goodTest.pas

< programsym >
< idsym , goodtestprogram >
[1] program GoodTestProgram;

< semicolonsym >
[2]     const

< constsym >
********ENTERING constant_definition_part********
< idsym , myname >
********ENTERING constant_definition********
< eqsym >
< stringvalsym , Derek >
[3]         myname = 'Derek';

< semicolonsym >
< idsym , age >
********ENTERING constant_definition********
< commasym >
< idsym , yearsalive >
< eqsym >
< integernosym , 27 >
[4]         age, yearsalive = 27;

< semicolonsym >
[5]     var

< varsym >
********ENTERING variable_definition********
< idsym , x >
********ENTERING variable_definition********
< commasym >
< idsym , y >
< commasym >
< idsym , z >
< colonsym >
< integersym >
[6]         x, y, z : integer;

< semicolonsym >
< idsym , a >
********ENTERING variable_definition********
< commasym >
< idsym , b >
< commasym >
< idsym , c >
```

```
50 < colonsym >
51 < charsym >
52 [7]         a, b, c : char;
53
54 < semicolonsym >
55 < idsym , f >
56 ********ENTERING variable_definition********
57 < colonsym >
58 < realsym >
59 [8]         f : real;
60
61 < semicolonsym >
62 < proceduresym >
63 ********ENTERING procedure_and_function_definition********
64 < idsym , testprocedure >
65 ********ENTERING procedure_declaration********
66 < lparensym >
67 < idsym , c1 >
68 ********ENTERING formal_parameters********
69 < commasym >
70 < idsym , c2 >
71 < colonsym >
72 < integersym >
73 < rparensym >
74 [10]    procedure testProcedure (c1,c2:integer);
75
76 < semicolonsym >
77 [11]        var
78
79 < varsym >
80 ********ENTERING constant_definition_part********
81 ********ENTERING variable_definition********
82 < idsym , c >
83 ********ENTERING variable_definition********
84 < colonsym >
85 < integersym >
86 [12]            c : integer;
87
88 < semicolonsym >
89 [13]        begin
90
91 < beginsym >
92 ********ENTERING procedure_and_function_definition********
93 ********ENTERING statement_part********
94 < idsym , c >
95 ********ENTERED statement********
96 < assignsym >
97 < idsym , c1 >
98 ********ENTERED assign********
99 ********ENTERED expression********
```

```
100 ********ENTERED simple_expression********
101 ********ENTERED term********
102 ********ENTERED factor********
103 [14]            c := c1;
104
105 < semicolonsym >
106 < idsym , c1 >
107 ********ENTERED statement********
108 < assignsym >
109 < idsym , c2 >
110 ********ENTERED assign********
111 ********ENTERED expression********
112 ********ENTERED simple_expression********
113 ********ENTERED term********
114 ********ENTERED factor********
115 [15]            c1 := c2;
116
117 < semicolonsym >
118 < idsym , c2 >
119 ********ENTERED statement********
120 < assignsym >
121 < idsym , c >
122 ********ENTERED assign********
123 ********ENTERED expression********
124 ********ENTERED simple_expression********
125 ********ENTERED term********
126 ********ENTERED factor********
127 [16]            c2 := c;
128
129 < semicolonsym >
130 < endsym >
131 [17]        end;
132
133 < semicolonsym >
134 < functionsym >
135 ********ENTERING procedure_and_function_definition********
136 < idsym , testfunction >
137 ********ENTERING function_declaration********
138 < lparensym >
139 < idsym , c1 >
140 ********ENTERING formal_parameters********
141 < commasym >
142 < idsym , c2 >
143 < colonsym >
144 < integersym >
145 < rparensym >
146 < colonsym >
147 < integersym >
148 [19]    function testFunction (c1,c2:integer) : integer;
149
```

```
150 < semicolonsym >
151 [20]        var
152
153 < varsym >
154 ********ENTERING constant_definition_part********
155 ********ENTERING variable_definition********
156 < idsym , c >
157 ********ENTERING variable_definition********
158 < colonsym >
159 < integersym >
160 [21]            c : integer;
161
162 < semicolonsym >
163 [22]        begin
164
165 < beginsym >
166 ********ENTERING procedure_and_function_definition********
167 ********ENTERING statement_part********
168 < idsym , c >
169 ********ENTERED statement********
170 < assignsym >
171 < idsym , c1 >
172 ********ENTERED assign********
173 ********ENTERED expression********
174 ********ENTERED simple_expression********
175 ********ENTERED term********
176 ********ENTERED factor********
177 < addsym >
178 < idsym , c2 >
179 ********ENTERED term********
180 ********ENTERED factor********
181 [23]            c := c1+c2;
182
183 < semicolonsym >
184 < endsym >
185 [24]        end;
186
187 < semicolonsym >
188 < proceduresym >
189 ********ENTERING procedure_and_function_definition********
190 < idsym , testprocedure2 >
191 ********ENTERING procedure_declaration********
192 < lparensym >
193 < idsym , c1 >
194 ********ENTERING formal_parameters********
195 < commasym >
196 < idsym , c2 >
197 < colonsym >
198 < integersym >
199 < rparensym >
```

```
200 [26]    procedure testProcedure2 (c1,c2:integer);
201
202 < semicolonsym >
203 [27]        var
204
205 < varsym >
206 ********ENTERING constant_definition_part********
207 ********ENTERING variable_definition********
208 < idsym , c >
209 ********ENTERING variable_definition********
210 < colonsym >
211 < integersym >
212 [28]            c : integer;
213
214 < semicolonsym >
215 [29]        begin
216
217 < beginsym >
218 ********ENTERING procedure_and_function_definition********
219 ********ENTERING statement_part********
220 < idsym , c >
221 ********ENTERED statement********
222 < assignsym >
223 < idsym , c1 >
224 ********ENTERED assign********
225 ********ENTERED expression********
226 ********ENTERED simple_expression********
227 ********ENTERED term********
228 ********ENTERED factor********
229 [30]            c := c1;
230
231 < semicolonsym >
232 < idsym , c1 >
233 ********ENTERED statement********
234 < assignsym >
235 < idsym , c2 >
236 ********ENTERED assign********
237 ********ENTERED expression********
238 ********ENTERED simple_expression********
239 ********ENTERED term********
240 ********ENTERED factor********
241 [31]            c1 := c2;
242
243 < semicolonsym >
244 < idsym , c2 >
245 ********ENTERED statement********
246 < assignsym >
247 < idsym , c >
248 ********ENTERED assign********
249 ********ENTERED expression********
```

```
250 ********ENTERED simple_expression********
251 ********ENTERED term********
252 ********ENTERED factor********
253 [32]            c2 := c;
254
255 < semicolonsym >
256 < endsym >
257 [33]        end;
258
259 < semicolonsym >
260 [34] begin
261
262 [35]    // This line will be skipped
263
264 < beginsym >
265 ********ENTERING procedure_and_function_definition********
266 ********ENTERING statement_part********
267 < writesym >
268 ********ENTERED statement********
269 < lparensym >
270 ********ENTERED write********
271 < stringvalsym , Enter a number:  >
272 ********ENTERED expression********
273 ********ENTERED simple_expression********
274 ********ENTERED term********
275 ********ENTERED factor********
276 < rparensym >
277 [36]    write('Enter a number: ');
278
279 < semicolonsym >
280 < readsym >
281 ********ENTERED statement********
282 < lparensym >
283 ********ENTERED read********
284 < idsym , x >
285 < rparensym >
286 [37]    read(x);
287
288 < semicolonsym >
289 < writesym >
290 ********ENTERED statement********
291 < lparensym >
292 ********ENTERED write********
293 < stringvalsym , You entered:  >
294 ********ENTERED expression********
295 ********ENTERED simple_expression********
296 ********ENTERED term********
297 ********ENTERED factor********
298 < rparensym >
299 [38]    write('You entered: ');
```

```
300
301 < semicolonsym >
302 < writelnsym >
303 ********ENTERED statement********
304 < lparensym >
305 ********ENTERED write********
306 < idsym , x >
307 ********ENTERED expression********
308 ********ENTERED simple_expression********
309 ********ENTERED term********
310 ********ENTERED factor********
311 < rparensym >
312 [39]    writeln(x);
313
314 [41]    // Test some real numbers
315
316 < semicolonsym >
317 < idsym , f >
318 ********ENTERED statement********
319 < assignsym >
320 < realnosym , 15.300000 >
321 ********ENTERED assign********
322 ********ENTERED expression********
323 ********ENTERED simple_expression********
324 ********ENTERED term********
325 ********ENTERED factor********
326 [42]    f := 15.3;
327
328 < semicolonsym >
329 < idsym , f >
330 ********ENTERED statement********
331 < assignsym >
332 < realnosym , 0.000000 >
333 ********ENTERED assign********
334 ********ENTERED expression********
335 ********ENTERED simple_expression********
336 ********ENTERED term********
337 ********ENTERED factor********
338 [43]    f := 3.25e-15;
339
340 < semicolonsym >
341 < idsym , f >
342 ********ENTERED statement********
343 < assignsym >
344 < realnosym , 1927000000.000000 >
345 ********ENTERED assign********
346 ********ENTERED expression********
347 ********ENTERED simple_expression********
348 ********ENTERED term********
349 ********ENTERED factor********
```

```
350 [44]    f := 19.27e+8;
351
352 < semicolonsym >
353 < idsym , f >
354 ********ENTERED statement********
355 < assignsym >
356 < realnosym , 552000000000.000000 >
357 ********ENTERED assign********
358 ********ENTERED expression********
359 ********ENTERED simple_expression********
360 ********ENTERED term********
361 ********ENTERED factor********
362 [45]    f := 55.2e10;
363
364 < semicolonsym >
365 < idsym , z >
366 ********ENTERED statement********
367 < assignsym >
368 < integernosym , 0 >
369 ********ENTERED assign********
370 ********ENTERED expression********
371 ********ENTERED simple_expression********
372 ********ENTERED term********
373 ********ENTERED factor********
374 [47]    z := 0;
375
376 < semicolonsym >
377 < whilesym >
378 ********ENTERED statement********
379 < lparensym >
380 ********ENTERED while********
381 ********ENTERED expression********
382 ********ENTERED simple_expression********
383 ********ENTERED term********
384 ********ENTERED factor********
385 < idsym , z >
386 ********ENTERED expression********
387 ********ENTERED simple_expression********
388 ********ENTERED term********
389 ********ENTERED factor********
390 < ltsym >
391 < idsym , x >
392 ********ENTERED simple_expression********
393 ********ENTERED term********
394 ********ENTERED factor********
395 < rparensym >
396 [48]    while (z < x) do
397
398 < dosym >
399 [49]        begin
```

```
400
401 < beginsym >
402 ********ENTERING statement_part********
403 < writesym >
404 ********ENTERED statement********
405 < lparensym >
406 ********ENTERED write********
407 < idsym , z >
408 ********ENTERED expression********
409 ********ENTERED simple_expression********
410 ********ENTERED term********
411 ********ENTERED factor********
412 < rparensym >
413 < semicolonsym >
414 < writesym >
415 ********ENTERED statement********
416 < lparensym >
417 ********ENTERED write********
418 < stringvalsym , ,  >
419 ********ENTERED expression********
420 ********ENTERED simple_expression********
421 ********ENTERED term********
422 ********ENTERED factor********
423 < rparensym >
424 [50]            write(z); write(', ');
425
426 < semicolonsym >
427 < idsym , z >
428 ********ENTERED statement********
429 < assignsym >
430 < idsym , z >
431 ********ENTERED assign********
432 ********ENTERED expression********
433 ********ENTERED simple_expression********
434 ********ENTERED term********
435 ********ENTERED factor********
436 < addsym >
437 < integernosym , 1 >
438 ********ENTERED term********
439 ********ENTERED factor********
440 [51]            z := z + 1;
441
442 < semicolonsym >
443 < endsym >
444 [52]        end;
445
446 < semicolonsym >
447 < writelnsym >
448 ********ENTERED statement********
449 < lparensym >
```

```
450 ********ENTERED write********
451 < idsym , z >
452 ********ENTERED expression********
453 ********ENTERED simple_expression********
454 ********ENTERED term********
455 ********ENTERED factor********
456 < rparensym >
457 [53]    writeln(z);
458
459 < semicolonsym >
460 < writelnsym >
461 ********ENTERED statement********
462 < lparensym >
463 ********ENTERED write********
464 < ordsym >
465 ********ENTERED expression********
466 ********ENTERED simple_expression********
467 ********ENTERED term********
468 ********ENTERED factor********
469 < lparensym >
470 ********ENTERING ord********
471 < charvalsym , 0 >
472 ********ENTERED expression********
473 ********ENTERED simple_expression********
474 ********ENTERED term********
475 ********ENTERED factor********
476 < rparensym >
477 < rparensym >
478 [55]    writeln(ord('0')); // Rest of this line skipped
479
480 < semicolonsym >
481 < idsym , y >
482 ********ENTERED statement********
483 < assignsym >
484 < idsym , z >
485 ********ENTERED assign********
486 ********ENTERED expression********
487 ********ENTERED simple_expression********
488 ********ENTERED term********
489 ********ENTERED factor********
490 < multsym >
491 < lparensym >
492 ********ENTERED factor********
493 < idsym , z >
494 ********ENTERED expression********
495 ********ENTERED simple_expression********
496 ********ENTERED term********
497 ********ENTERED factor********
498 < addsym >
499 < ordsym >
```

```
500 ********ENTERED term********
501 ********ENTERED factor********
502 < lparensym >
503 ********ENTERING ord********
504 < charvalsym , 0 >
505 ********ENTERED expression********
506 ********ENTERED simple_expression********
507 ********ENTERED term********
508 ********ENTERED factor********
509 < rparensym >
510 < minussym >
511 < integernosym , 3 >
512 ********ENTERED term********
513 ********ENTERED factor********
514 < rparensym >
515 < addsym >
516 < idsym , x >
517 ********ENTERED term********
518 ********ENTERED factor********
519 < idivsym >
520 < integernosym , 2 >
521 ********ENTERED factor********
522 [56]    y := z * (z + (* inline comment *) ord('0') - 3
    ) + x div 2;
523
524 < semicolonsym >
525 < writelnsym >
526 ********ENTERED statement********
527 < lparensym >
528 ********ENTERED write********
529 < idsym , y >
530 ********ENTERED expression********
531 ********ENTERED simple_expression********
532 ********ENTERED term********
533 ********ENTERED factor********
534 < rparensym >
535 [57]    writeln(y);
536
537 < semicolonsym >
538 < idsym , z >
539 ********ENTERED statement********
540 < assignsym >
541 < idsym , testfunction >
542 ********ENTERED assign********
543 ********ENTERED expression********
544 ********ENTERED simple_expression********
545 ********ENTERED term********
546 ********ENTERED factor********
547 < lparensym >
548 < idsym , x >
```

```
549 ********ENTERED application********
550 ********ENTERED expression********
551 ********ENTERED simple_expression********
552 ********ENTERED term********
553 ********ENTERED factor********
554 < commasym >
555 < idsym , y >
556 ********ENTERED expression********
557 ********ENTERED simple_expression********
558 ********ENTERED term********
559 ********ENTERED factor********
560 < rparensym >
561 < addsym >
562 < idsym , x >
563 ********ENTERED term********
564 ********ENTERED factor********
565 [59]    z := testFunction(x, y) + x;
566
567 [63]    { Muliline
568     comment
569     }
570
571 < semicolonsym >
572 < ifsym >
573 ********ENTERED statement********
574 < lparensym >
575 ********ENTERED if********
576 ********ENTERED expression********
577 ********ENTERED simple_expression********
578 ********ENTERED term********
579 ********ENTERED factor********
580 < idsym , x >
581 ********ENTERED expression********
582 ********ENTERED simple_expression********
583 ********ENTERED term********
584 ********ENTERED factor********
585 < gtsym >
586 < idsym , y >
587 ********ENTERED simple_expression********
588 ********ENTERED term********
589 ********ENTERED factor********
590 < rparensym >
591 [64]    if (x > y) then
592
593 < thensym >
594 [65]        begin
595
596 < beginsym >
597 ********ENTERING statement_part********
598 < writelnsym >
```

```
599 ********ENTERED statement********
600 < lparensym >
601 ********ENTERED write********
602 < stringvalsym , x is bigger than y! >
603 ********ENTERED expression********
604 ********ENTERED simple_expression********
605 ********ENTERED term********
606 ********ENTERED factor********
607 < rparensym >
608 [66]            writeln('x is bigger than y!');
609
610 < semicolonsym >
611 [67]        end
612
613 < endsym >
614 [68]    else
615
616 < elsesym >
617 [69]        begin
618
619 < beginsym >
620 ********ENTERING statement_part********
621 < writelnsym >
622 ********ENTERED statement********
623 < lparensym >
624 ********ENTERED write********
625 < stringvalsym , x is smaller than y! >
626 ********ENTERED expression********
627 ********ENTERED simple_expression********
628 ********ENTERED term********
629 ********ENTERED factor********
630 < rparensym >
631 [70]            writeln('x is smaller than y!');
632
633 < semicolonsym >
634 < endsym >
635 [71]        end;
636
637 [76]    (* Another
638     multline
639     comment
640     *)
641
642 < semicolonsym >
643 < ifsym >
644 ********ENTERED statement********
645 < lparensym >
646 ********ENTERED if********
647 ********ENTERED expression********
648 ********ENTERED simple_expression********
```

```
649 ********ENTERED term********
650 ********ENTERED factor********
651 < idsym , z >
652 ********ENTERED expression********
653 ********ENTERED simple_expression********
654 ********ENTERED term********
655 ********ENTERED factor********
656 < eqsym >
657 < idsym , x >
658 ********ENTERED simple_expression********
659 ********ENTERED term********
660 ********ENTERED factor********
661 < rparensym >
662 [77]    if (z = x) then
663
664 < thensym >
665 [78]        begin
666
667 < beginsym >
668 ********ENTERING statement_part********
669 < writelnsym >
670 ********ENTERED statement********
671 < lparensym >
672 ********ENTERED write********
673 < stringvalsym , z is equal to x >
674 ********ENTERED expression********
675 ********ENTERED simple_expression********
676 ********ENTERED term********
677 ********ENTERED factor********
678 < rparensym >
679 [79]            writeln('z is equal to x');
680
681 < semicolonsym >
682 < endsym >
683 [80]        end;
684
685 < semicolonsym >
686 < idsym , x >
687 ********ENTERED statement********
688 < assignsym >
689 < integernosym , 65 >
690 ********ENTERED assign********
691 ********ENTERED expression********
692 ********ENTERED simple_expression********
693 ********ENTERED term********
694 ********ENTERED factor********
695 [82]    x := 65;
696
697 < semicolonsym >
698 < whilesym >
```

```
699 ********ENTERED statement********
700 < lparensym >
701 ********ENTERED while********
702 ********ENTERED expression********
703 ********ENTERED simple_expression********
704 ********ENTERED term********
705 ********ENTERED factor********
706 < idsym , x >
707 ********ENTERED expression********
708 ********ENTERED simple_expression********
709 ********ENTERED term********
710 ********ENTERED factor********
711 < ltsym >
712 < integernosym , 90 >
713 ********ENTERED simple_expression********
714 ********ENTERED term********
715 ********ENTERED factor********
716 < rparensym >
717 [84]    while (x < 90) do
718
719 < dosym >
720 [85]        begin
721
722 < beginsym >
723 ********ENTERING statement_part********
724 < writesym >
725 ********ENTERED statement********
726 < lparensym >
727 ********ENTERED write********
728 < chrsym >
729 ********ENTERED expression********
730 ********ENTERED simple_expression********
731 ********ENTERED term********
732 ********ENTERED factor********
733 < lparensym >
734 ********ENTERING chr********
735 < idsym , x >
736 ********ENTERED expression********
737 ********ENTERED simple_expression********
738 ********ENTERED term********
739 ********ENTERED factor********
740 < rparensym >
741 < rparensym >
742 < semicolonsym >
743 < writesym >
744 ********ENTERED statement********
745 < lparensym >
746 ********ENTERED write********
747 < stringvalsym , ,  >
748 ********ENTERED expression********
```

```
749 ********ENTERED simple_expression********
750 ********ENTERED term********
751 ********ENTERED factor********
752 < rparensym >
753 [86]            write({ another inline comment}chr(x));
    write(', ');
754
755 < semicolonsym >
756 < idsym , x >
757 ********ENTERED statement********
758 < assignsym >
759 < idsym , x >
760 ********ENTERED assign********
761 ********ENTERED expression********
762 ********ENTERED simple_expression********
763 ********ENTERED term********
764 ********ENTERED factor********
765 < addsym >
766 < integernosym , 1 >
767 ********ENTERED term********
768 ********ENTERED factor********
769 [87]            x := x + 1;
770
771 < semicolonsym >
772 < endsym >
773 [88]        end;
774
775 < semicolonsym >
776 < writelnsym >
777 ********ENTERED statement********
778 < lparensym >
779 ********ENTERED write********
780 < chrsym >
781 ********ENTERED expression********
782 ********ENTERED simple_expression********
783 ********ENTERED term********
784 ********ENTERED factor********
785 < lparensym >
786 ********ENTERING chr********
787 < idsym , x >
788 ********ENTERED expression********
789 ********ENTERED simple_expression********
790 ********ENTERED term********
791 ********ENTERED factor********
792 < rparensym >
793 < rparensym >
794 [89]    writeln(chr(x));
795
796 < semicolonsym >
797 < endsym >
```

```
798 [90] end.
799 < dotsym >
800 [90]
801
802 PARSING COMPLETED SUCCESSFULLY!!!!!
803
804 Saved astfp.txt
805
806 ===== SYMBOL TABLE =====
807 testprocedure2 (procedure) : 26 : NULL
808     c (integer) : 28 : 0
809     c2 (integer) : 26 : 0
810     c1 (integer) : 26 : 0
811 testfunction (function) : 19 : NULL
812     c (integer) : 21 : 0
813     c2 (integer) : 19 : 0
814     c1 (integer) : 19 : 0
815 testprocedure (procedure) : 10 : NULL
816     c (integer) : 12 : 0
817     c2 (integer) : 10 : 0
818     c1 (integer) : 10 : 0
819 f (real) : 8 : 0.000000
820 c (char) : 7 :
821 b (char) : 7 :
822 a (char) : 7 :
823 z (integer) : 6 : 0
824 y (integer) : 6 : 0
825 x (integer) : 6 : 0
826 yearsalive (integer) : 4 : 27
827 age (integer) : 4 : 27
828 myname (string) : 3 : Derek
829 goodtestprogram (program) : 1 : NULL
830 writeln (keyword) : 0 : NULL
831 write (keyword) : 0 : NULL
832 readln (keyword) : 0 : NULL
833 read (keyword) : 0 : NULL
834 ord (keyword) : 0 : NULL
835 chr (keyword) : 0 : NULL
836 const (keyword) : 0 : NULL
837 var (keyword) : 0 : NULL
838 real (keyword) : 0 : NULL
839 integer (keyword) : 0 : NULL
840 string (keyword) : 0 : NULL
841 char (keyword) : 0 : NULL
842 of (keyword) : 0 : NULL
843 array (keyword) : 0 : NULL
844 while (keyword) : 0 : NULL
845 do (keyword) : 0 : NULL
846 then (keyword) : 0 : NULL
847 else (keyword) : 0 : NULL
```

```
848 if (keyword) : 0 : NULL
849 not (keyword) : 0 : NULL
850 or (keyword) : 0 : NULL
851 and (keyword) : 0 : NULL
852 end (keyword) : 0 : NULL
853 begin (keyword) : 0 : NULL
854 function (keyword) : 0 : NULL
855 procedure (keyword) : 0 : NULL
856 program (keyword) : 0 : NULL
857 mod (keyword) : 0 : NULL
858 div (keyword) : 0 : NULL
859 derektrom@Raelyns-MBP mini-pascal-compiler-V2 %
860
```